DESIGN AND IMPLEMENTATION OF A UNIFIED HARDWARE ARCHITECTURE FOR CRYPTOGRAPHIC HASH PRIMITIVES

THESIS

Submitted in partial fulfillment of the requirements of

BITS C422T Thesis

By

ROHIT KOUL
2002A7TS041

Under the supervision of

T S B Sudarshan
Assistant Professor
CS – IS Group

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)

06th, May 2006

DESIGN AND IMPLEMENTATION OF A UNIFIED HARDWARE ARCHITECTURE FOR CRYPTOGRAPHIC HASH PRIMITIVES

THESIS

Submitted in partial fulfillment of the requirements of

BITS C422T Thesis

By

ROHIT KOUL
2002A7TS041

Under the supervision of

T S B Sudarshan
Assistant Professor
CS – IS Group

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)
06th May, 2006

ACKNOWLEDGEMENTS

I wish to express my gratitude to Prof. T S B Sudarshan for giving me an opportunity to pursue this Thesis work under his supervision, and also for being a constant source of inspiration and guidance throughout the course of this work.

I also sincerely thank Dr. S. Gurunarayanan and Mr. Pawan Sharma for providing me with access to the excellent facilities available at the Oyster-Lab for the purpose of this work.

I am also grateful to Ganesh T.S. and Rakesh K for the numerous doubt-clearing mails and chats.

I think I will defeat the sole purpose of this acknowledgements exercise if I fail to thank my friends especially Deepika , Rajesh , Suraj , Ashish and Chakri for their support and cooperation during the entire semester. You'll never know how many different problems I solved and concepts I built and re-built after I had taken a break to hang out with you for a couple of hours.

CERTIFICATE

This is to certify that the Thesis titled, '**Design and Implementation of a Unified Hardware Architecture for Cryptographic Hash Primitives**', submitted by **Rohit Koul**, ID No. **2002A7TS041**, in partial fulfillment of the requirements of BITS C422T Thesis, embodies the work done by him under my supervision.

(Supervisor)
T S B Sudarshan
Assistant Professor
CS – IS Group

6th May, 2006

LIST OF SYMBOLS AND ABBREVIATIONS

FPGA	Field Programmable Gate Array
HDLs	Hardware Description Languages
MD-5	Message Digest, Version 5
SHA-256	Secure Hash Algorithm, Version 2
RIPEMD-160	RACE Integrity Primitives Evaluation Message Digest, 160 Bit
SKC	Secret Key Cryptography
PKC	Public Key Cryptography
IV	Initial Value
CV	Chaining Variables
PROM	Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
EEPROM	Electrically Erasable Programmable Read Only Memory
SRAM	Static Random Access Memory
PLA	Programmable Logic Array
PLD	Programmable Logic Devices
FPD	Field Programmable Devices
MPGA	Masked Programmable Gate Array
CMOS	Complementary Metal Oxide Semiconductor
OTP	One Time Programmable
RP	Reprogrammable
CAD	Computer Aided Design
SPLD	Simple Programmable Logic Devices
CPLD	Complex Programmable Logic Devices
VHDL	Very High Speed Integrated Circuit Hardware Description Language
Verilog	Verifying Logic
ASIC	Application Specific Integrated Circuits
RISC	Reduced Instruction Set Computing
CFSM	Cryptographic Finite State Machine
RTL	Register Transfer Level

Thesis Title	Design and Implementation of a Unified Hardware Architecture for Cryptographic Hash Primitives		
Supervisor			T S B Sudarshan
Semester	Second	**Session**	2005 – 2006
Name of Student	Rohit Koul	**ID No.**	2002A7TS041

Abstract

With the increasing prominence of the Internet as a tool of commerce, security has become a tremendously important issue. One essential aspect for secure communication over networks is that of cryptography. It ensures that the authentication, privacy, integrity and non-repudiation aspects of communication are not compromised. The increasing prominence of mobile devices has increased the necessity for hardware implementations of cryptography algorithms. There are three classes of cryptography algorithms, secret key, public key and one way hash. Hash Algorithms are a class of cryptographic primitives used for fulfilling the requirements of integrity and authentication in cryptography. This thesis aims at proposing and implementing a unified architecture for several popular one way hash algorithms such as MD-5, SHA-256, RIPEMD-160 and Tiger on a programmable logic device commensurable in complexity to FPGAs.

TABLE OF CONTENTS

Chapter I Introduction

The Internet provides essential communication between millions of people and is being increasingly used as a tool of commerce. In this context, security becomes a tremendously important issue to deal with .The Telegraph, telephone, radio, and especially the computer have put everyone on the globe within earshot----at the price of our privacy. It may feel like we are performing an intimate act when, sequestered in our rooms and cubicles, we casually use our cell phones and computers to transmit our thoughts, confidences, business plans, and even our money. But clever eavesdroppers and sometimes even not-so-clever ones can hear it all. We think we are whispering, but we really are broadcasting!!!!

A potential antidote exists: “Cryptography”, the use of secret codes and ciphers to scramble information so that it's worthless to anyone but the intended recipients. And it is through the magic of cryptography that many communication conventions of the real world--such as signatures, contracts, receipts, and even poker games--have found their way to the ubiquitous electronic commons. There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of cryptography. With increasing prominence of communications on mobile devices, the software implementations of the various algorithms are found to be disadvantageous in terms of speed, power consumption and other such factors. Therefore, the necessity for hardware implementation of cryptography algorithms has increased manifold. The details of the cryptographic algorithms are presented in Chapter II.

The applications of cryptographic hardware include, but are not restricted to, Internet routers, electronic financial transactions, remote access servers, virtual private networks, mobile telephone networks and satellite communications. Implementing cryptography algorithms on programmable hardware gives us a host of advantages like algorithm agility, algorithm uploading, algorithm modification, architecture efficiency and cost efficiency, in addition to being reconfigurable. Reconfigurable Computing and FPGAs are covered in Chapter III.

The present status of research work in designing cryptography algorithms on hardware is

presented in Chapter IV. Chapter V explains in detail the proposed unified architecture for the four hash algorithms. Designing cryptographic algorithms on hardware is basically a digital system design process. Therefore, coverage of digital system design guidelines as well as coding for synthesizability in Verilog has been done. Chapter VI concludes the report with appropriate guidelines for future work.

Appendix A gives details of the VLSI CAD Tools used along with the project file listing while Appendix B explains the guidelines for simulation and shows certain screen shots of the Simulation.

2.1 Basics of Cryptography

Cryptography is the science of writing in secret code and is an ancient art; the first documented use of cryptography in writing dates back to circa 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription. Some experts argue that cryptography appeared spontaneously sometime after writing was invented, with applications ranging from diplomatic missives to war-time battle plans. It is no surprise, then, that new forms of cryptography came soon after the widespread development of computer communications. In data and telecommunications, cryptography is necessary when communicating over any non trusted medium, which includes just about *any* network, particularly the Internet.

Within the context of any application-to-application communication, there are some specific security requirements, including:

- *Authentication:* The process of proving one's identity. (The primary forms of host-to-host authentication on the Internet today are name-based or address-based, both of which are notoriously weak.)
- *Privacy/confidentiality:* Ensuring that no one can read the message except the intended receiver.
- *Integrity:* Assuring the receiver that the received message has not been altered in any way from the original.
- *Non-repudiation:* A mechanism to prove that the sender really sent this message.

Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication.

There are, in general, three types of cryptographic schemes typically used to accomplish these goals: secret key (or symmetric) cryptography, public-key (or asymmetric) cryptography, and hash functions, each of which is described below. In all cases, the initial unencrypted data is referred to as *plaintext*. It is encrypted into *ciphertext*, which will in turn (usually) be decrypted into usable plaintext.

2.2 Types of Cryptographic Algorithms

There are several ways of classifying cryptographic algorithms. For purposes of this thesis, the categorization will be based on the number of keys that are employed for encryption and decryption. The three types of algorithms that will be considered are (Figure 2.1):

- Secret Key Cryptography: Uses a single key for both encryption and decryption
- Public Key Cryptography: Uses one key for encryption and another for decryption
- Hash Functions: Uses a mathematical transformation to irreversibly "encrypt" information

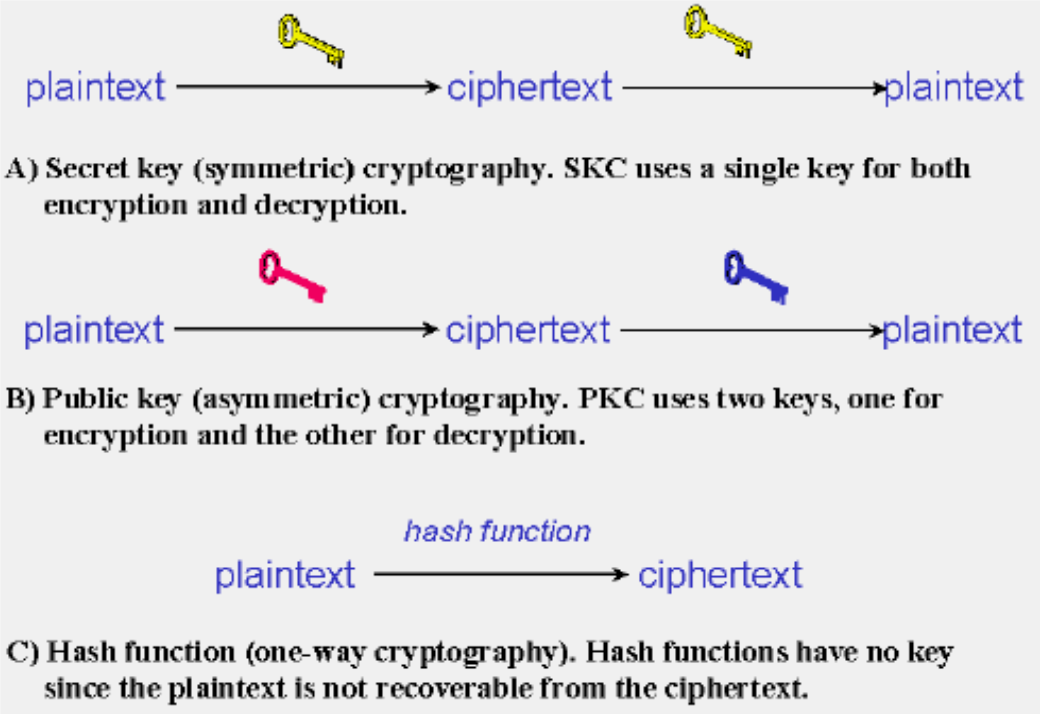


Fig. 2.1 Three Classes of Cryptography Algorithms

In *secret key cryptography*, a single key is used for both encryption and decryption. As shown in Fig. 2.1A, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or rule set) to decrypt the message and recover the plaintext. Because a single key is used for both functions, secret key cryptography is also called *symmetric encryption*. There are several widely used secret key cryptography schemes and they are generally categorized as being

either *stream ciphers* or *block ciphers*. Examples of secret key cryptography algorithms include DES, IDEA, RC5, Blowfish etc.

Public-key cryptography (PKC) has been said to be the most significant new development in cryptography in the last 300-400 years. It was originally conceptualized as a two-key crypto system in which two parties could engage in a secure communication over a non-secure communications channel without having to share a secret key. key is used to encrypt the plaintext and the other key is used to decrypt the ciphertext. The important point here is that it does not matter which key is applied first, but that both keys are required for the process to work (Fig. 2.1B). Because a pair of keys is required, this approach is also called *asymmetric cryptography*. In PKC, one of the keys is designated the *public key* and may be advertised as widely as the owner wants. The other key is designated the *private key* and is never revealed to another party. The most common example of PKC is the RSA algorithm.

Hash functions, also called *message digests* and *one-way encryption*, are algorithms that, in some sense, use no key (Fig. 2.1C). Instead, a fixed-length hash value is computed based upon the plaintext that makes it impossible for either the contents or length of the plaintext to be recovered. Furthermore, there is an almost zero probability that two different plaintext messages will yield the same hash value. Hash algorithms are typically used to provide a *digital fingerprint* of a file's contents often used to ensure that the file has not been altered by an intruder or virus. Hash functions are also commonly employed by many operating systems to encrypt passwords. Examples of popular one way hash algorithms include MD-2, MD-4, MD-5, SHA-1, RIPEMD-160 etc.

2.3 Cryptography Algorithms in Practice

Each cryptography scheme is optimized for some specific application(s). Hash functions, for example, are well-suited for ensuring data integrity because any change made to the contents of a message will result in the receiver calculating a different hash value than the one placed in the transmission by the sender. Since it is highly unlikely that two different messages will yield the same hash value, data integrity is ensured to a high degree of confidence. Secret-key cryptography, on the other hand, is ideally suited to encrypting messages. The sender can generate a *session key* on a per-message basis to

encrypt the message; the receiver, of course, needs the same session key to decrypt the message. Key exchange, of course, is a key application of public-key cryptography. Asymmetric schemes can also be used for non-repudiation; if the receiver can obtain the session key encrypted with the sender's private key, then only this sender could have sent the message. Public-key cryptography could, theoretically, also be used to encrypt messages although this is rarely done because secret-key cryptography operates about 1000 times faster than public-key cryptography.

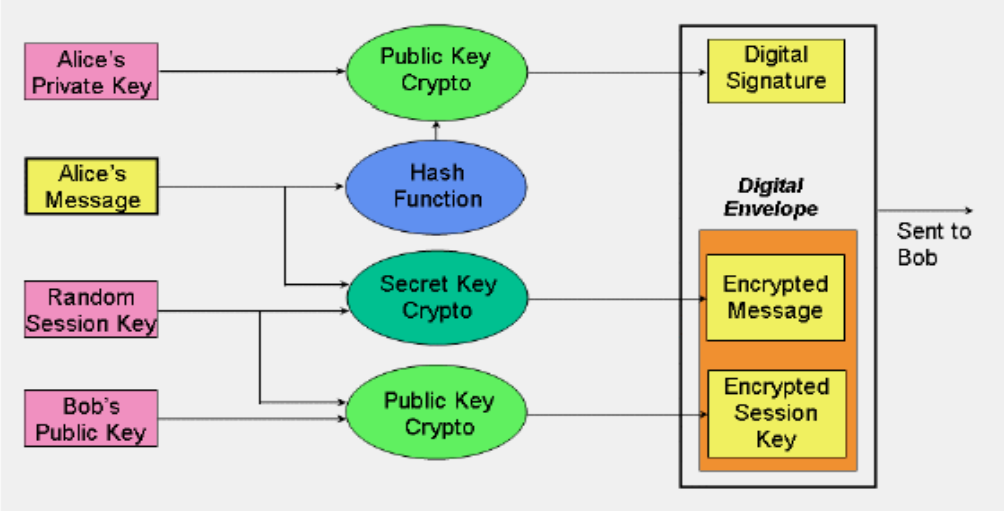


Fig. 2.2 Cryptography Algorithms in Practice

Fig. 2.2 puts all of this together and shows how a *hybrid cryptographic* scheme combines all of these functions to form a secure transmission comprising *digital signature* and *digital envelope*. In this example, the sender of the message is Alice and the receiver is Bob. A digital envelope comprises an encrypted message and an encrypted session key. Alice uses secret key cryptography to encrypt her message using the *session key*, which she generates at random with each session. Alice then encrypts the session key using Bob's public key. The encrypted message and encrypted session key together form the digital envelope. Upon receipt, Bob recovers the session secret key using his private key and then decrypts the encrypted message. The digital signature is formed in two steps. First, Alice computes the hash value of her message; next, she encrypts the hash value with her private key. Upon receipt of the digital signature, Bob recovers the hash value calculated by Alice by decrypting the digital signature with Alice's public key. Bob can then apply the hash function to Alice's original message, which he has already decrypted. If the resultant hash value is not the same as the value supplied by Alice, then Bob

knows that the message has been altered; if the hash values are the same, Bob should believe that the message he received is identical to the one that Alice sent. This scheme also provides non-repudiation since it proves that Alice sent the message; if the hash value recovered by Bob using Alice's public key proves that the message has not been altered, then only Alice could have created the digital signature. Bob also has proof that he is the intended receiver; if he can correctly decrypt the message, then he must have correctly decrypted the session key meaning that his is the correct private key. More details can be had from [1].

The rest of the report will solely concentrate on the third class of cryptography algorithms, namely, one way hash functions.

2.4 One Way Hash Algorithms

Cryptographic hash functions play a fundamental role in modern cryptography. While related to conventional hash functions commonly used in non-cryptographic computer applications – in both cases, larger domains are mapped to smaller ranges – they differ in several important aspects. Our focus is restricted to cryptographic hash functions (hereafter, simply hash functions), and in particular to their use for data integrity and message authentication.

Hash functions take a message as input and produce an output referred to as a *hashcode*, *hash-result*, *hash-value*, or simply *hash*. More precisely, a hash function h maps bitstrings of arbitrary finite length to strings of fixed length, say n bits. For a domain D and range R with $h : D \rightarrow R$ and $|D| > |R|$, the function is many-to-one, implying that the existence of *collisions* (pairs of inputs with identical output) is unavoidable. Indeed, restricting h to a domain of t-bit inputs ($t > n$), if h were "random" in the sense that all outputs were essentially equiprobable, then about 2^{t-n} inputs would map to each output, and two randomly chosen inputs would yield the same output with probability 2^{-n} (independent of t).

The basic idea of cryptographic hash functions is that a hash-value serves as a compact representative image (sometimes called an *imprint*, *digital fingerprint*, or *message*

digest) of an input string, and can be used as if it were uniquely identifiable with that string. For further mathematical treatment of the one way hash functions, reference can be made to [2].

2.4.1 General Model for Iterated Hash Functions

Most unkeyed hash functions h are designed as iterative processes which hash arbitrary length inputs by processing successive fixed-size blocks of the input, as illustrated in Fig. 2.3

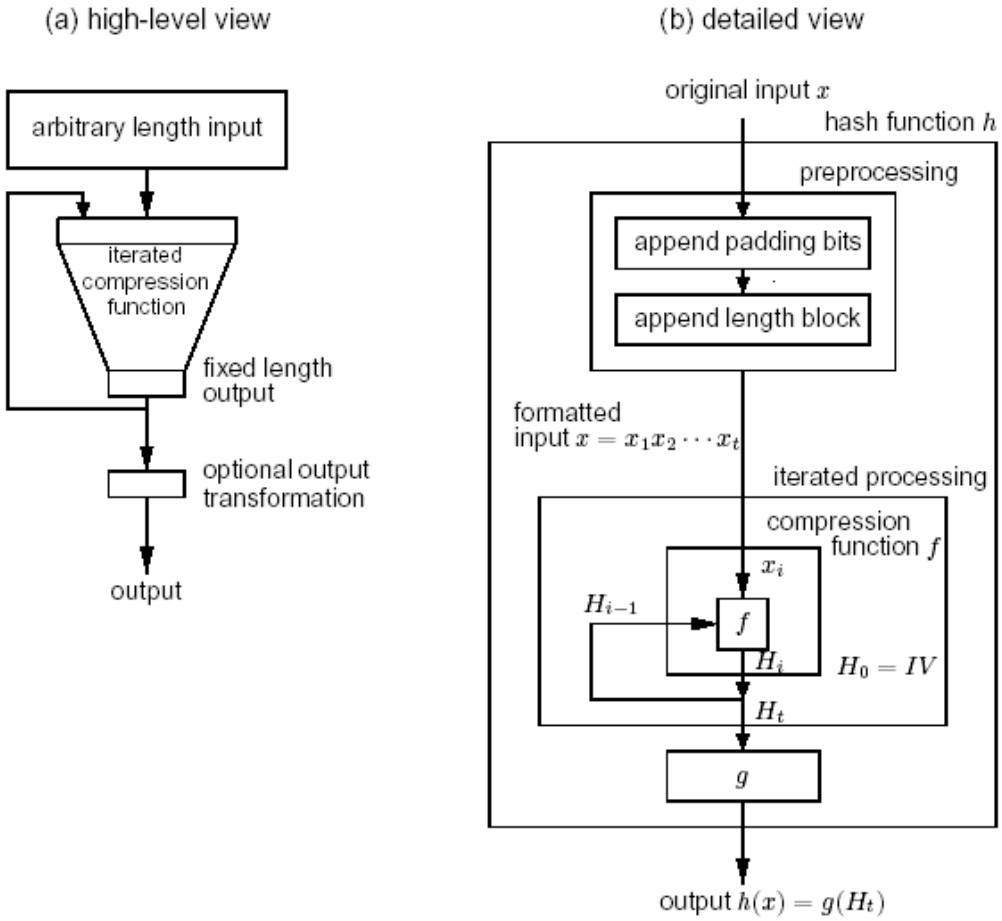


Fig. 2.3 General Model for an Iterated Hash Function

A hash input x of arbitrary finite length is divided into fixed-length r-bit blocks x_i. This preprocessing typically involves appending extra bits (*padding*) as necessary to attain an overall bit length which is a multiple of the block length r, and often includes (for

security reasons) a block or partial block indicating the bit length of the unpadded input. Each block x_i then serves as input to an internal fixed-size hash function f, the *compression function* of h, which computes a new intermediate result of bit length n for some fixed n, as a function of the previous n-bit intermediate result and the next input block x_i. Letting H_i denote the partial result after stage i, the general process for an iterated hash function with input $x = x_1x_2\ldots\ldots x_t$ can be modeled as follows:

$$H_0 = IV\ ;\ H_i = f(H_{i-1};\ x_i);\ 1 \leq i \leq t;\ h(x) = g(H_t)$$

H_{i-1} serves as the n-bit *chaining variable* between stage i - 1 and stage i, and H_0 is a pre-defined starting value or *initializing value* (IV). An optional output transformation g (see Fig. 2.3) is used in a final step to map the n-bit chaining variable to an m-bit result $g(H_t)$; g is often the identity mapping $g(H_t) = H_t$.

Particular hash functions are distinguished by the nature of the preprocessing, compression function, and output transformation. In this thesis, three major one way hash algorithms are considered, namely, MD-5, SHA-1 and RIPEMD-160. The details of each are presented below.

2.4.2 Message Digest, Version 5 (MD-5)

MD5 [3] is a message digest algorithm developed by Ron Rivest at MIT. It is basically a secure version of his previous algorithm, MD4 which is a little faster than MD5. This has been the most widely used secure hash algorithm particularly in Internet-standard message authentication. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit message digest of the input. This is mainly intended for digital signature applications where a large file must be compressed in a secure manner before being encrypted with a private (secret) key under a public key cryptosystem.

Assume that there is an arbitrarily large message as input and that its message digest is to be determined. The processing involves the following steps.

(1) Padding

The message is padded to ensure that its length in bits plus 64 is divisible by 512. That is, its length is congruent to 448 modulo 512. Padding is always performed even if the length of the message is already congruent to 448 modulo 512. Padding consists of a single 1-bit followed by the necessary number of 0-bits.

(2) Appending length

A 64-bit binary representation of the original length of the message is concatenated to the result of step (1). (Least Significant Byte first). The expanded message at this level will exactly be a multiple of 512-bits. Let the expanded message be represented as a sequence of L 512-bit blocks $Y_0, Y_1,..,Y_q,..,Y_{L-1}$ as shown in Fig. 2.4 [4]. Note that in the figure, IV and CV represent initial value and chaining variable respectively.

(3) Initialize the MD buffer

The variables IV and CV are represented by a four–word buffer (ABCD) used to compute the message digest. Here each A, B, C, D is a 32-bit register and they are initialized as IV to the following values in hexadecimal. Low-order bytes are put first.

Word A: 01 23 45 67

Word B: 89 AB CD EF

Word C: FE DC BA 98

Word D: 76 54 32 10

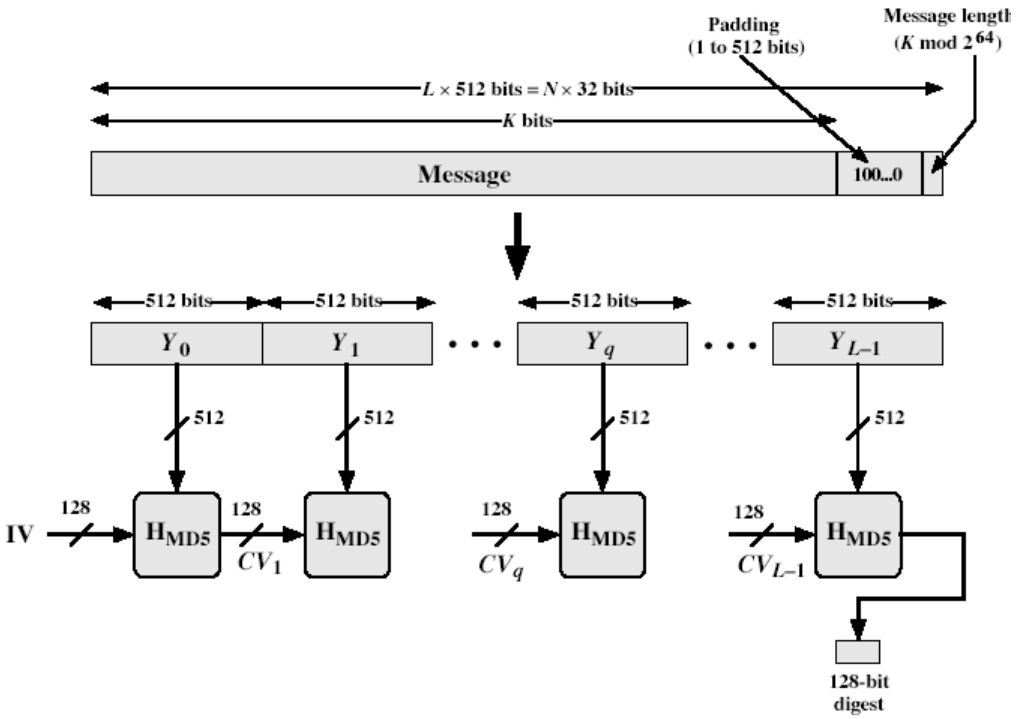


Fig. 2.4 Message Digest Generation using MD-5

(4) Process message in 16-word blocks

This is the heart of the algorithm, which includes four "rounds" of processing. It is represented by H_{MD5} in Fig. 2.4 and its logic is given in Fig. 2.5. The four rounds have similar structure but each uses different auxiliary functions *F, G, H* and *I*.

$$F(X, Y, Z) = (X \square Y) \square (X' \square Y)$$

$$G(X, Y, Z) = (X \square Z) \square (Y \square Z')$$

$$H(X, Y, Z) = X \square Y \square Z$$

$$I(X, Y, Z) = Y \square (X \square Z')$$

where □, □, □ and ‘ represent the logical OR, AND, XOR and NOT operations, respectively. Each round consists of 16 steps and each step uses a 64-element table T [1 ... 64] constructed from the sine function. Let $T[i]$ denote the i-th element of the table, which is equal to the integer part of 2^{32} *abs(sin(i)), where i is in radians. Each round also takes as input the current 512-bit block (Yq) and the 128-bit chaining variable (CV_q). An array X of 32-bit words holds the current 512-bit Y_q. For the first round the words are used in their original order. The following permutations of the words are defined for rounds 2 through 4:

$$\square_2(i) = (1+ 5i) \bmod 16$$

$$\square_3(i) = (5+ 3i) \bmod 16$$

$$\square_4(i) = 7i \bmod 16$$

The output of the fourth round is added to the input of the first round (CV_q) to produce CV_{q+1}.

(5) Output

After all L 512-bit blocks have been processed, the output from Lth stage is the 128-bit message digest. Fig. 2.6 shows the operations involved in a single step. The additions are modulo 2^{32}. Four different circular shift amounts

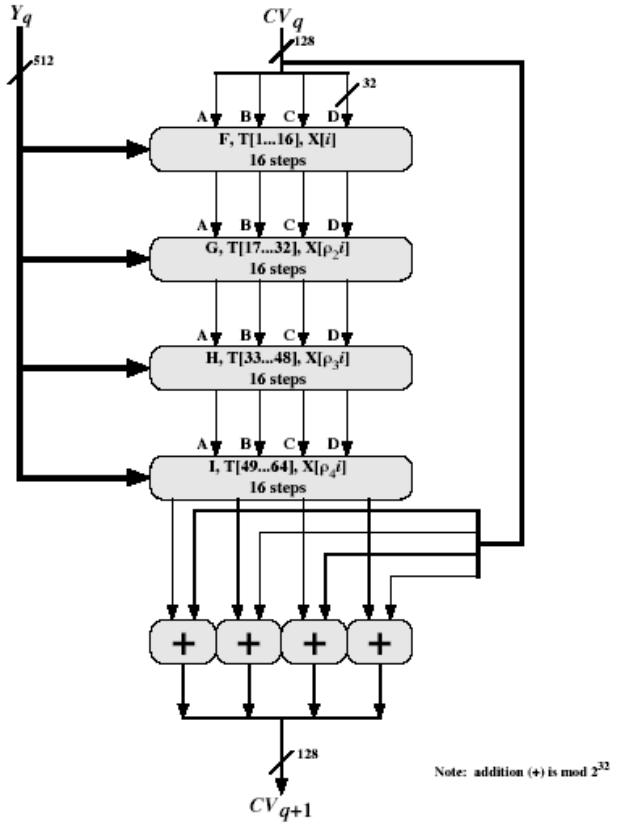


Fig. 2.5 H_{MD5} – The MD5 Compression Function

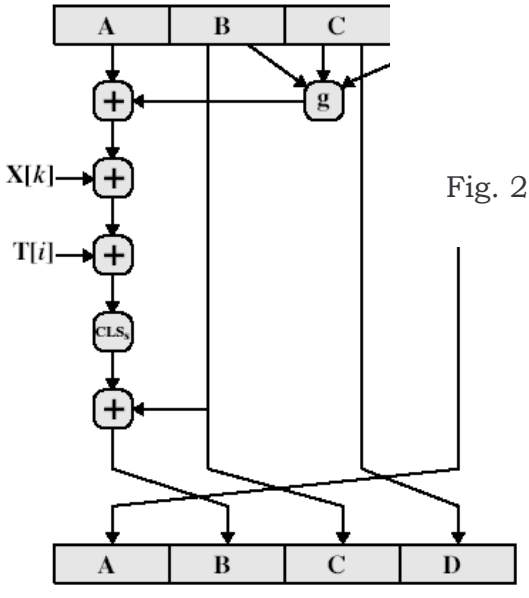


Fig. 2.6 Elementary MD5 Operation (Single Step)

(s) are used each round and are different from round to round. Each step is of the following form [4]:

$$A \leftarrow D$$

$$B \leftarrow B + ((A + g(B, C, D) + X[k] + T[i]) << s)$$

$$C \leftarrow B$$

$$D \leftarrow C$$

Each of the 64 32 bit word elements of T is used exactly once, during one step of one round. For each step, only one of the 4 bytes of the ABCD buffer is updated. Thus, each byte gets updated four times during the round and then a final time at the end to produce the final output for this block. Four different circular left shift amounts are used each round, and they are different from round to round. The point of all this complexity is to make it very difficult to generate collisions (two 512-bit blocks that produce the same output).

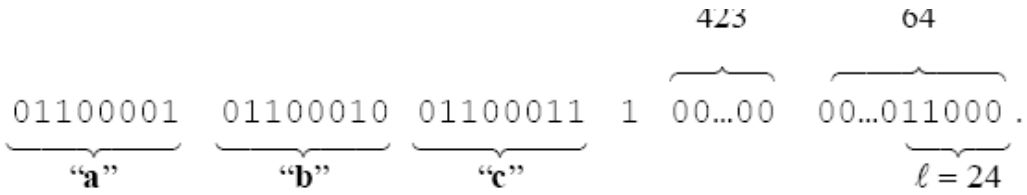
2.4.3 Secure Hash Algorithm, Version 2 (SHA-256)

Ever Since certain attacks have been reported on SHA-1 exploiting its two fundamental weaknesses (One is that the file preprocessing step is not complicated enough; another is that certain math operations in the first 20 rounds have unexpected security problems) the scientific community is looking forward to adopting SHA-2 as the *de-facto* standard. NIST (National Institute of Standards and Technology) published four additional hash functions in the SHA family, each with longer digests, collectively known as SHA-2. The individual variants are named after their digest lengths (in bits): "SHA-256", "SHA-384", and "SHA-512". They were first published in 2001 in the draft FIPS PUB 180-2, at which time review and comment were accepted. SHA-256 and SHA-512 are novel hash functions computed with 32- and 64-bit words, respectively. They use different shift amounts and additive constants, but their structures are otherwise virtually identical, differing only in the number of rounds. SHA-224 and SHA-384 are simply truncated versions of the first two, computed with different initial values.

Preprocessing :

Suppose that the length of the message, *M*, is l bits. Append the bit “1” to the end of the

message, followed by *k* zero bits, where *k* is the smallest, non-negative solution to the equation l +1 + k = 512 mod 448 . Then append the 64-bit block that is equal to the number l expressed using a binary representation. For example, the (8-bit ASCII) message "**abc**" has length 24 =3 x 8 , so the message is padded with a one bit, then 448-(24-1)= 423 zero bits, and then the message length, to become the 512-bit padded message.



The length of the padded message now becomes a multiple of 512

Parsing the message :

For SHA-1 and SHA-256, the padded message is parsed into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

Setting the initial hash value :

the initial hash value, $H(0)$, shall consist of the following eight 32-bit words, in hex:

Ho = 6a09e667 *H1* = bb67ae85 *H2* = 3c6ef372 *H3* = a54ff53a

H4 = 510e527f *H5* = 9b05688c *H6* = 1f83d9ab *H7* = 5be0cd19.

These words are obtained by taking the first 64 bits of the fractional parts of the square roots of the first eight prime numbers.

The Computation:

Process message in 16-word blocks

This is the heart of the algorithm, which includes four "rounds" of processing. Its logic is given in Fig. 2.7. The four rounds have similar structure but each uses different functions

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$$
$$\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$$
$$\sigma_0^{\{512\}}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x)$$
$$\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)$$

Fig 2.7 The Functions used in SHA-256

where □, □, □ and ‘ represent the logical OR, AND, XOR and NOT operations, respectively. Each round consists of 16 steps and each round uses a 4-element table Kt [1 ... 4]. Each round also takes as input the current 512-bit block (*Yq*) and the 256-bit chaining variable (CV_q). An array *X* of 32-bit words holds the current 512-bit Y_q. The output of the fourth round is added to the input of the first round (CV_q) to produce CV_{q+1}.

(5) Output

After all *L* 512-bit blocks have been processed, the output from *L*th stage is the 256-bit message digest. Fig. 2.8 shows the operations involved in a single step. The additions are modulo 2^{32}. Each step is of the following form [4]:

T1=h+Σ(e)+Ch(e,f,g)+Kt+Wt

T2=Σ(a)+Maj(a,b,c)

(a,b,c,d,e,f,g,h)=(T1+T2,a,b,c,d+T1,e,f,g)

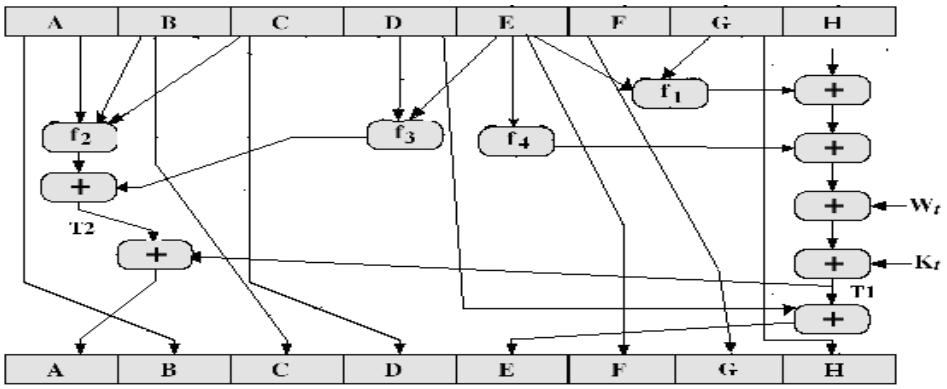


Fig. 2.8 Elementary SHA 256 Operation (Single Step)

It remains to be indicated how the 32 bit word values W_t are derived from the 512 bit message. Fig. 2.9 illustrates the mapping. The first sixteen values are taken directly from the message block, and the remaining values are determined using the following definition

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

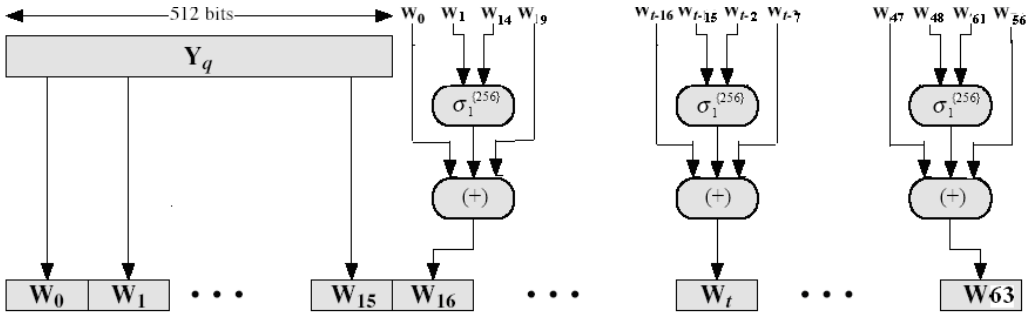


Fig. 2.8 Creation of 64 Word Input Sequence for SHA-256 Processing of a Single Block

, Unlike MD5 and RIPEMD160, SHA-256 expands the 16 block words to 64 words for use in the compression function. This introduces a great deal of redundancy and interdependence into the message blocks, making the probability of finding a collision very less.

2.4.4 RACE Integrity Primitives Evaluation Message Digest-160 (RIPEMD-160)

RIPEMD-160 (RACE Integrity Primitives Evaluation Message Digest) [6] is a 160-bit message digest algorithm (and cryptographic hash function) developed in Europe by Hans Dobbertin, Antoon Bosselaers and Bart Preneel, and first published in 1996. It is an improved version of RIPEMD, which in turn was based upon the design principles used in MD4, and is similar in both strength and performance to the more popular SHA-1.

Assume that there is an arbitrarily large message as input and that its message digest is to be determined. The processing involves the following steps

(1) Padding

The message is padded to ensure that its length in bits plus 64 is divisible by 512. That is, its length is congruent to 448 modulo 512. Padding is always performed even if the length of the message is already congruent to 448 modulo 512. Padding consists of a single 1-bit followed by the necessary number of 0-bits.

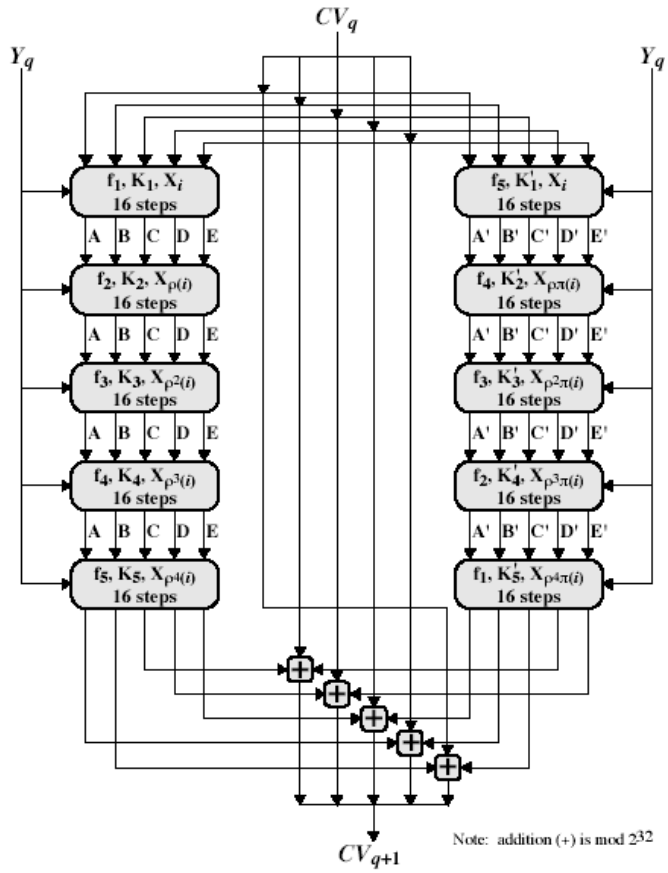
(2) Appending length

A 64-bit binary representation of the original length of the message is concatenated to the result of step (1). (Least Significant Byte first). The expanded message at this level will exactly be a multiple of 512-bits. Let the expanded message be represented as a sequence of L 512-bit blocks $Y_0, Y_1,..,Y_q,..,Y_{L-1}$

(3) Initialize the MD buffer

The variables IV and CV are represented by a five–word buffer (ABCDE) used to compute the message digest. Here each A, B, C, D, E is a 32-bit register and they are initialized as IV to the following values in hexadecimal. These words are the same as the initial values for MD5, but they are stored in Big Endian format.

Word A: 01 23 45 67



Word B: 89 AB CD EF

Word C: FE DC BA 98

Word D: 76 54 32 10

Word E: F0 E1 D2 C3

(4) Process message in 16-word blocks

This is the heart of the algorithm, which includes four "rounds" of processing. Its logic is given in Fig. 2.10. The ten rounds have similar structure but each uses different auxiliary functions *f_1, f_2, f_3, f_4 and f_5.*

$F_1 (X,Y,Z)=X \square Y \square Z$

Fig. 2.10 RIPEMD-160 Processing of a Single 512 Bit Block

$$F_2(X,Y,Z)=(X \square Y) \square (X' \square Z)$$

$$F_3(X,Y,Z)=(X \square Y') \square Z$$

$$F_4(X,Y,Z)=(X \square \square) \square (Y \square Z')$$

$$F_5(X,Y,Z)=X \square (Y \square Z')$$

where □, □, □ and ‘ represent the logical OR, AND, XOR and NOT operations, respectively. Each round consists of 16 steps and each round uses a different constant. Two rounds can run in parallel as shown in Fig. 2.10. Each round also takes as input the current 512-bit block (Yq) and the 160-bit chaining variable (CV_q). An array X of 32-bit words holds the current 512-bit Y_q. The output of the two parallel blocks’ fifth round is added to the input of the first round (CV_q) as shown in Fig. 2.10 to produce CV_{q+1}.

(5) Output

After all L 512-bit blocks have been processed, the output from Lth stage is the 160-bit message digest. Fig. 2.11 shows the operations involved in a single step. The additions are modulo 2^{32}. Each step is of the following form [4]:

$$A \leftarrow E$$

$$B \leftarrow E + ((A + f(t,B,C,D) + X[i] + K[j]) << s)$$

$$C \leftarrow B$$

$$D \leftarrow S^{10}(C)$$

$$E \leftarrow D$$

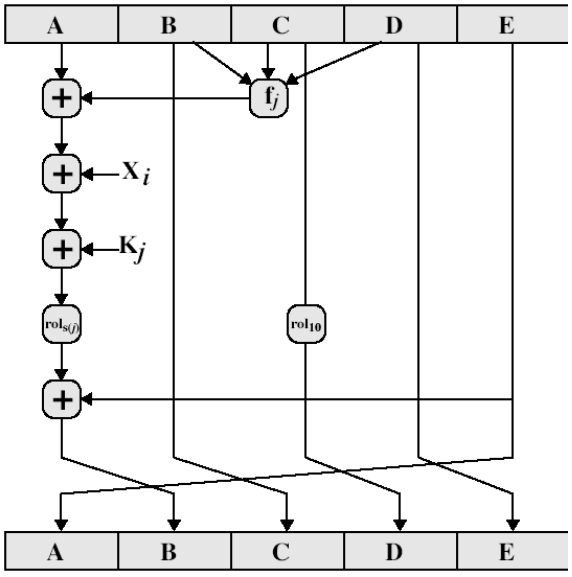


Fig. 2.11 Elementary RIPEMD-160 Operation (Single Step)

For details on the shift amounts and the message word to select for each step in each round, reference can be made to [4]. Two parallel lines of 5 rounds each increase the complexity of finding collisions between rounds. The two parallel rounds, though of the same structure, have some fundamental differences, which make them more secure. These include the difference in the functions and additive constants used, as well as the order of processing of the words in the block.

2.4.5 Tiger Algorithm

Cryptographic hash functions are very important for cryptographic protocols. Some hash functions are based on feed forward modes of block ciphers, but the main contenders have been the functions based on MD4, which include MD5, RIPE-MD, SHA and SHA-1. Another family was Snefru, and its derivative Snefru-8. However, collisions for Snefru were found in 1990, and recently a collision of MD4 has also been found. These attacks cast doubt on the security of the other members of these families. From the performance point of view, all the functions mentioned above were designed for 32-bit processors. The next generation of processors has 64-bit words, and includes the DEC Alpha series as well as forthcoming processors from Intel, HP and IBM. From these considerations, it was believed that a next generation hash function:

- should be secure. At the very least it must be one-way, collision-free and multiplication-free;
- should run quickly on 64-bit processors, and yet not run too slowly on the already fielded 32-bit machines such as Intel's 80486;
- should, insofar as possible, be usable as a drop-in replacement for MD4, MD5, SHA and SHA-1.

Therefore, **Tiger** a fast new hash function was proposed. It is designed to be very fast on modern computers, and in particular on the state-of-the-art 64-bit computers (like DEC-Alpha), while it is still not slower than other suggested hash functions on 32-bit machines. On DEC-Alpha, Tiger hashes more than 132Mbits per second (measured on Alpha 7000, Model 660, on one processor). On the same machine, MD5 hashes only about 37Mbps. On 32-bit machines, the code of Tiger is not fully optimized. Still it hashes faster than MD5 on 486s and Pentiums. Tiger has no usage restrictions nor patents. It can be used freely, with the reference implementation, with other implementations or with a modification to the reference implementation (as long as it still implements Tiger).

Its main operation is table lookup into four S-boxes, each from eight bits to 64 bits. The other operations are 64-bit additions and subtractions, 64-bit multiplication by small constants (5, 7 and 9), 64-bit shifts and logical operations such as XOR and NOT. For

drop-in compatibility, we adopt the outer structure of the MD4 family: the message is padded by a single `1' bit followed by a string of `0's and finally the message length as a 64-bit word. The result is divided into n 512-bit blocks. The result is divided into n 512-bit blocks.

The size of the hash value, and of the intermediate state, is three words, or 192 bits. This value was chosen for the following reasons:

1. Since we use 64-bit words, the size should be a multiple of 64;
2. To be compatible with applications using SHA-1, the hash size should be at least 160 bits;
3. All the successful shortcut attacks on existing hash functions attack the intermediate state, rather than the final hash value. The attacker typically chooses two colliding values for an intermediate block, and this propagates to a collision of the full function. However, these attacks would not work if the intermediate hash values were larger.

Tiger with the full 192 bits of output in use may be called Tiger/192. When replacing other functions in existing applications, we suggest two shorter variants:

1. Tiger/160: the hash value is the first 160 bits of the result of Tiger/192, and is used for compatibility with SHA and SHA-1;
2. *Tiger/128: the hash value is the first 128 bits of the result of Tiger/192, and is used for compatibility with MD4, MD5, RIPE-MD.*

The efficiency of this function is partially based on the potential parallelism in its design. In each round of Tiger, the eight table lookup operations can be done in parallel, so compilers can make best use of pipelining. The design also allows efficient hardware implementation. The memory size required by Tiger is only slightly more than the size of the four S boxes. If this can be accommodated within the cache of the processor, the computation runs about twice as fast (measured on DEC Alpha). The size of the four S boxes is $4 \cdot 256 \cdot 8 = 8096 = 8$Kbytes, which is about the size of the cache on most machines. If eight S boxes were used, 16 Kbytes would be required, which is twice as the size of the cache on Alpha.

2.4.5.1 Working of Tiger Algorithm

In Tiger all the computations are on 64-bit words, in little-endian/2-complement representation. We use three 64-bit registers called a, b, and c as the intermediate hash values. These registers are initialized to h_0 which is:

a = 0x0123456789ABCDEF

b = 0xFEDCBA9876543210

c = 0xF096A5B4C3B2E187

Each successive 512-bit message block is divided into eight 64-bit words x0, x1, ..., x7, and the following computation is performed to update h_i to h_{i+1}.

This computation consists of three passes, and between each of them there is a *key schedule* --- an invertible transformation of the input data which prevents an attacker forcing sparse inputs in all three rounds. Finally there is a feedforward stage in which the new values of a, b, and c are combined with their initial values to give h_{i+1}.

The steps of the tiger algorithm are:

1) save_abc

2) pass(a,b,c,5)

3) key_schedule

4) pass(c,a,b,7)

5) key_schedule

6) pass(b,c,a,9)

7) feedforward

where

1) save_abc saves the value of h_i

```
aa = a ;
bb = b ;
cc = c ;
```

2) pass(a,b,c,mul) is

```
round(a,b,c,x0,mul);
round(b,c,a,x1,mul);
```

```
round(c,a,b,x2,mul);
round(a,b,c,x3,mul);
round(b,c,a,x4,mul);
round(c,a,b,x5,mul);
round(a,b,c,x6,mul);
round(b,c,a,x7,mul);
```

where round(a,b,c,x,mul) is

```
c ^= x ;
a -= t1[c_0] ^ t2[c_2] ^ t3[c_4] ^ t4[c_6] ;
b += t4[c_1] ^ t3[c_3] ^ t2[c_5] ^ t1[c_7] ;
b *= mul;
```

and where c_i is the **i**th byte of **c** ($0 \le i \le 7$).

3) key_schedule is

```
x0 -= x7 ^ 0xA5A5A5A5A5A5A5A5;
x1 ^= x0;
x2 += x1;
x3 -= x2 ^ ((~x1)<<19);
x4 ^= x3;
x5 += x4;
x6 -= x5 ^ ((~x4)>>23);
x7 ^= x6;
x0 += x7;
x1 -= x0 ^ ((~x7)<<19);
x2 ^= x1;
x3 += x2;
x4 -= x3 ^ ((~x2)>>23);
x5 ^= x4;
x6 += x5;
x7 -= x6 ^ 0x0123456789ABCDEF;
```

where << and >> are logical (rather than arithmetic) shift left and shift right operators.

4) feedforward is

a ^= aa ;

b -= bb ;

c += cc ;

The resultant registers a, b, c are the 192 bits of the (intermediate) hash value h_{i+1}.

Tiger is designed to be both fast and secure. Its core is three rounds, each of which uses eight lookups into 8-to-64-bit S-boxes to provide a strong nonlinear avalanche plus a number of register operations to increase diffusion and make differential attacks harder. It can be implemented efficiently on 32-bit and 64-bit machines.

2.4.6 Comparing MD-5, SHA-256 , RIPEMD-160 and Tiger

In terms of resistance to brute force attacks [4], all algorithms are invulnerable with regard to weak collision resistance. At 128 bits, MD-5 is highly vulnerable to a birthday attack [4], whereas both SHA-256 and RIPEMD-160 at 160 bits each are safe in the foreseeable future.

All shortcut attacks on MD*/ Snefru target one of the intermediate blocks. In Tiger Increasing the intermediate value to 192 bits helps thwart these attacks.

The key schedule ensures that changing a small number of bits in a message affects many bits during the various passes. Together with the strong avalanche, it helps Tiger to resist attacks similar to Dobbertin's differential attack on MD4.

The multiplication of the register b in each round also contributes to the resistance to such attacks, since it ensures that bits which were used as inputs to S boxes in the previous rounds are mixed into other S boxes as well, and to the same S boxes with a different input difference. This multiplication also prevents related-key attacks on the hash function, since the constant differs in each round.

A lot of progress has been made in the cryptanalysis of MD5. RIPEMD-160 is expected to fare better in comparison to SHA-256, even though the latter appears to be highly resistant to cryptanalytic attacks.

Due to the added complexity of SHA-256 and RIPEMD-160, they are slower in operational speed in comparison to MD-5. In Tiger There is a strong avalanche, in that each message bit affects all the three registers after three rounds --- much faster than in any other hash function. The avalanche in 64-bit words (and 64-bit S boxes) is much faster than when shorter words are used.

MD5 and RIPEMD-160 use Little Endian Scheme while SHA-256 and its other variants use Big Endian Scheme for interpreting a message as a sequence of 32 bit words.

3. Introduction to Reconfigurable Computing and Field Programmable Gate arrays

3.1 Reconfigurable Computing

There are two primary methods in traditional computing for the execution of algorithms. The first is to use an Application Specific Integrated Circuit, or ASIC, to perform the operations in hardware. Because these ASICs are designed specifically to perform a given computation, they are very fast and efficient when executing the exact computation for which they were designed. However, after fabrication the circuit cannot be altered. Microprocessors are a far more flexible solution. Processors execute a set of instructions to perform a computation. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance suffers, and is far below that of an ASIC. The processor must read each instruction from memory, determine its meaning, and only then execute it. This results in a high execution overhead for each individual operation. Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware.

3.1.1 Reconfigurable Computing Architectures

There are many different architectures designed for use in reconfigurable computing. One of the primary variations between these is the degree of coupling (if any) with a host microprocessor. Programmable logic tends to be inefficient at implementing certain

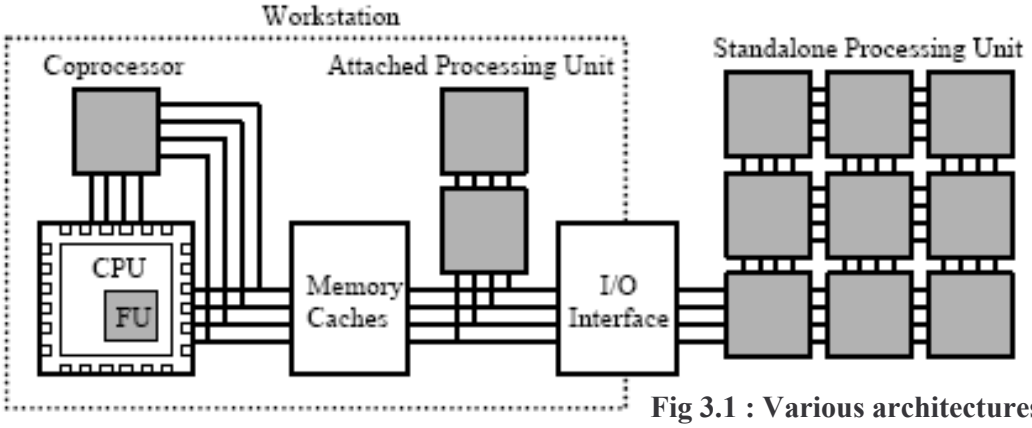


Fig 3.1 : Various architectures

types of operations, such as loop and branch control**.** *In order to most efficiently run an application in a reconfigurable computing system, the areas of the program that cannot be easily mapped to the reconfigurable logic are executed on a host microprocessor. Meanwhile, the areas with a high density of computation that can benefit from implementation in hardware are mapped to the reconfigurable logic. For the systems that use a microprocessor in conjunction with reconfigurable logic, there are several ways in which these two computation structures may be coupled (see Figure 1).*

First, reconfigurable hardware can be used solely to provide reconfigurable functional units within a host processor. This allows for a traditional programming environment with the addition of custom instructions that may change over time. Here, the reconfigurable units execute as functional units on the main microprocessor datapath, with registers used to hold the input and output operands.

Second, a reconfigurable unit may be used as a coprocessor. A coprocessor is in general larger than a functional unit, and is able to perform computations without the constant supervise on of the host processor. Instead, the processor initializes the reconfigurable hardware and either sends the necessary data to the logic, or provides information on where this data might be found in memory.

Third, an attached reconfigurable processing unit behaves as if it is an additional processor in a multiprocessor system. The host processor's data cache is not visible to the attached reconfigurable processing unit. There is, therefore, a higher delay in communication between the host processor and the reconfigurable hardware, such as when communicating configuration information, input data, and results. However, this type of reconfigurable hardware does allow for a great deal of computation independence, by shifting large chunks of a computation over to the reconfigurable hardware.

Finally, the most loosely coupled form of reconfigurable hardware is that of an external standalone processing unit. This type of reconfigurable hardware communicates infrequently with a host processor (if present). This model is similar to that of networked workstations, where processing may occur for very long periods of time without a great deal of communication.

3.1.2. Comparison and Applications

Each of these styles has distinct benefits and drawbacks. The tighter the integration of the reconfigurable hardware, the more frequently it can be used within an application or

set of applications due to a lower communication overhead. However, the hardware is unable to operate for significant portions of time without intervention from a host processor, and the amount of reconfigurable logic available is often quite limited. The more loosely coupled styles allow for greater parallelism in program execution, but suffer from higher communications overhead. In applications that require a great deal of communication, this can reduce or remove any acceleration benefits gained through this type of reconfigurable hardware. In addition to the level of coupling, the design of the actual computation blocks within the reconfigurable hardware varies from system to system. Each unit of computation, or logic block, can be as simple as a 3-input look up table (LUT), or as complex as a 4-bit ALU. This difference in block size is commonly referred to as **the granularity** of the logic block, where the 3-bit LUT is an example of a very fine grained computational element, and a 4-bit ALU is an example of a quite coarse grained unit. The finer grained blocks are useful for bit-level manipulations, while the coarse grained blocks are better optimized for standard datapath applications.

Very fine-grained logic blocks (such as those operating only on 2 or 3 one-bit values) are useful for bit level manipulation of data, as can frequently be found in encryption and image processing applications. Also, because the cells are fine grained, computation structures of arbitrary bit widths can be created. This can be useful for implementing datapath circuits that are based on data-widths not implemented on the host processor (5 bit multiply, 128 bit addition, etc). Performing these types of computation on a traditional microprocessor wastes calculation effort for the case of very small operands, and incurs multi-instruction overhead for the case of very large operands. The reconfigurable logic performs exactly the calculation that is needed.

Several reconfigurable systems use a medium-sized granularity of logic block. A number of these architectures operate on two or more 4-bit wide data words, in particular. This increases the total number of input lines to the circuit, and provides more efficient computational structures for more complex problems. Medium-grained logic blocks may be used to implement datapath circuits of varying bit widths, similar to the fine-grained structures. However, with the ability to perform more complex operations of a greater number of inputs, this type of structure can also be used to efficiently implement more complex operations such as finite state machines.

Very coarse-grained architectures are primarily intended for the implementation of word-width datapath circuits. Because the logic blocks used are optimized for large

computations, they will perform these operations much more quickly (and consume less chip area) than a set of smaller cells connected to form the same type of structure. However, because their composition is static, they are unable to leverage optimizations in the size of operands. For example, the RaPiD architecture [6], is composed of 16-bit adders, multipliers, and registers. If only three 1-bit values are required, then the use of this architecture suffers an unnecessary area and speed overhead, as all 16 bits are computed. However, these coarse grained architectures can be much more efficient than fine-grained architectures for implementing functions closer to their basic word size.

3.1.3 Routing in Reconfigurable Systems

The routing between the logic blocks within the reconfigurable hardware is also of great importance. Routing contributes significantly to the overall area of the reconfigurable hardware. Yet, when the percentage of logic blocks used in an FPGA becomes very high, automatic routing tools frequently have difficulty achieving the necessary connections between the blocks. Good routing structures are therefore essential to ensure that a design can be successfully placed and routed onto the reconfigurable hardware.

There are two primary methods to provide both local and global routing resource. The first is the use of segmented routing. In segmented routing, short wires accommodate local communications traffic. These short wires can be connected together using switchboxes to emulate longer wires. Optionally, longer wires may also be included, and signals may transfer between local and distance routing at connection blocks. Hierarchical routing provides local routing within a cluster, and longer wires at the boundaries connect the different clusters together. Hierarchical structures are optimized for situations where the most communication should be local and only a limited amount of communication will traverse long distances.

Reconfigurable systems that are composed of multiple FPGA chips interconnected on a single processing board have additional hardware concerns over single-chip systems. In particular, there is a need for an efficient connection scheme between the chips, as well as to external memory and the system bus. This is to provide for circuits that are too large to fit within a single FPGA, but may be partitioned over the multiple FPGAs available. A number of different interconnection schemes have been explored [2], including meshes, crossbars, and variants on these structures. Because of the need for efficient communication between the FPGAs, the determining the inter-chip routing topology is a very important step in the design of a multi-FPGA system. Once a circuit

has been configured onto the reconfigurable hardware, it is ready to be used by the host processor during program execution.

The runtime operation of a reconfigurable system occurs in two distinct phases: configuration and execution. The configuration of the reconfigurable hardware is under the control of the host processor. This host processor directs a stream of configuration data to the reconfigurable hardware, and this configuration data is used to define the actual operation of the hardware. Configurations can be loaded solely at startup of a program, or periodically during runtime, depending on the design of the system. More concepts involved in run-time reconfiguration (the dynamic reconfiguration of devices during computation execution) are discussed in a later section. The actual execution model of the reconfigurable hardware varies from system to system. Some systems suspend the execution of the host processor during execution on the reconfigurable hardware. Others allow for simultaneous execution with techniques similar to the use of fork/join primitives in multiprocessor programming.

3.2 Field Programmable Gate Arrays

In the mid 1980s a new technology for implementing digital logic was introduced, the field-programmable gate array (FPGA). These devices could either be viewed as small, slow mask programmable gate arrays (MPGAs) or large, expensive programmable logic devices (PLDs). FPGAs were capable of implementing significantly more logic than PLDs, especially because they could implement multi-level logic, while most PLDs were optimized for two-level logic. Although they did not have the capacity of MPGAs, they also did not have to be custom fabricated, greatly lowering the costs for low-volume parts, and avoiding long fabrication delays. While many of the FPGAs were configured by static RAM cells in the array (SRAM), this was generally viewed as a liability by potential customers who worried over the chip's volatility. Antifuse-based FPGAs also were developed, and for many applications were much more attractive, both because they tended to be smaller and faster due to less programming overhead, and also because there was no volatility to the configuration.

In the late 1980s and early 1990s there was a growing realization that the volatility of SRAM-based FPGAs was not a liability, but was in fact the key to many new types of applications. Since the programming of such an FPGA could be changed by a completely electrical process, much as a standard processor can be configured to run many programs, SRAM-based FPGAs have become the workhorse of many new

reprogrammable applications. Some of the most exciting new uses of FPGAs move beyond the implementation of digital logic, and instead harness large numbers of FPGAs as a general-purpose computation medium. The circuit mapped onto the FPGAs need not be standard hardware equations, but can even be operations from algorithms and general computations.

3.2.1 FPGA Technology

One of the most common field-programmable elements is programmable logic devices (PLDs). PLDs concentrate primarily on two-level, sum-of-products implementations of logic functions. They have simple routing structures with predictable delays. Since they are completely prefabricated, they are ready to use in seconds, avoiding long delays for chip fabrication. Field-Programmable Gate Arrays (FPGAs) are also completely prefabricated, but instead of two-level logic they are optimized for multi-level circuits. This allows them to handle much more complex circuits on a single chip, but it often sacrifices the predictable delays of PLDs. Just as in PLDs, FPGAs are completely prefabricated, and contain special features for customization. These configuration points are normally either SRAM cells, EPROM, EEPROM, or antifuses. Antifuses are one-time programmable devices , which when “blown” create a connection, while when “unblown” no current can flow between their terminals. Because the configuration of an antifuse is permanent, antifuse-based FPGAs are one-time programmable, while SRAM-based FPGAs are reprogrammable, even in the target system. Since SRAMs are volatile, an SRAM-based FPGA must be reprogrammed every time the system is powered up, usually from a ROM included in the circuit to hold configuration files. SRAM cells are larger than antifuses and EEPROM/EPROM, meaning that SRAM-based FPGAs will have fewer configuration points than FPGAs using other programming technologies. However, SRAM-based FPGAs have numerous advantages. Since they are easily reprogrammable, their configurations can be changed for bug fixes or upgrades. Thus they provide an ideal prototyping medium. Also, these devices can be used in situations where they can expect to have numerous different configurations, such as multi-mode systems and reconfigurable computing machines.

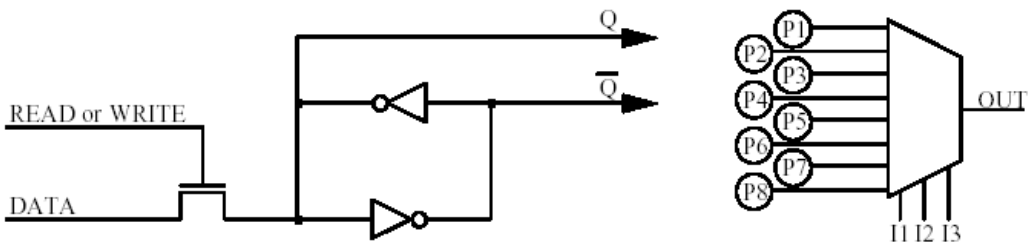


Fig.3.2 Programming bit for SRAM based FPGAs

In SRAM-based FPGAs memory cells are scattered throughout the FPGA. As shown in Figure 2, a pair of cross-coupled inverters will sustain whatever value is programmed onto them. A single n-transistor gate is provided for either writing a value or reading a value back out. The ratio of sizes between the transistor and the upper inverter is set to allow values sent through the n-transistor to overpower the inverter. The read back feature is used during debugging to determine the current state of the system. The actual control of the FPGA is handled by the *Q* and *Q* outputs. One simple application of an SRAM bit is to have the Q terminal connected to the gate of an n-transistor. If a 1 is assigned to the programming bit, the transistor is closed, and values can pass between the source and drain. If a 0 is assigned, the transistor is opened, and values cannot pass. Thus, this construct operates similarly to an antifuse, though it requires much more area. One of the most useful SRAM-based structures is the lookup table (LUT). By connecting 2N programming bits to a multiplexer (Figure 2), any N input combinational Boolean function can be implemented. Although it can require a large number of programming bits for large N, LUTs of up to 5 inputs can provide a flexible, powerful function implementation medium.

3.2.2 FPGA Architecture

Most commercial SRAM-based FPGA architectures have the same basic structure, a two-dimensional array of programmable logic blocks, that can implement a variety of bit-wise logic functions, surrounded by channels of wire segments to interconnect logic block I/O. In most cases, FPGA logic blocks contain one or more programmable lookup tables,that can be programmed to perform any Boolean logic function of a small number of inputs (typically 4-5), a small number of simple Boolean logic gates, and one or more

inputs-outputs. User-programmable switches control interconnection between adjacent wire segments and wire segments and logic blocks.

Three main classes of SRAM-based FPGA architecture have evolved over the past decade: cell-based, hierarchical, and island-style. Each architecture is defined by the amount of logic that can be implemented in an array logic block and the length and interconnection pattern of its channel wire segments.

Cell-based FPGA architectures, consist of a two-dimensional array of simple logic blocks which typically contain two or three two-input logic structures such as XOR, AND, and NAND gates. Inter-logic block communication is primarily made through direct-wired connections from block outputs to inputs on adjacent logic blocks. Small numbers of wire segments that span multiple logic blocks over a minimal amount of global communication but typically not enough to implement circuits with randomized communication patterns. These routing restrictions frequently limit the application domain of these devices to circuits with primarily nearest-neighbor connectivity such as bit-serial arithmetic units and regular2-D filter arrays.

Devices with a ***hierarchical*** architecture, contain a 2-D array of complex logic blocks with many lookup tables and inputs-outputs (typically 8 or more) per block. Inter-logic block signals are carried on wire segments that span the entire device providing numerous high-speed paths between device I/O and internal logic. This architectural choice leads to an ideal implementation setting for designs with many high-fanout signals. These devices can selectively be used to implement many types of logic circuits exhibiting a variety of interconnection patterns.

Island-style devices provide an architectural compromise between cell-based and hierarchical architectures. Island-style devices are characterized by logic blocks of moderate complexity generally containing a small number of lookup tables (typically 2-4) per block. Routing channels with a range of wire segment lengths are available to support both local and global device routing.

3.2.3 Reconfigurable Computers based on FPGAs

Reconfigurable computers based on FPGAs have shown impressive speedups for a number of computing applications by customizing the underlying logic of the computing platform to create exactly the hardware functionality required. Typically, due to the size of the circuit created to perform the computation, multiple FPGA devices are needed for design implementation. A number of recent projects have used hundreds of FPGA

devices in concert as a reconfigurable computing platform to solve computational challenges such as shortest-path search calculation, array sorting, FFT calculation, and special-purpose processor implementation. As the complexity of reconfigurable computing applications and target platforms grows, the ability of application designers to map designs by hand to reconfigurable hardware becomes limited by the amount of time needed to analyze the complex variety of hardware implementation trade offs available. These limitations have given rise to automated high-level design flows for multi-FPGA reconfigurable computing platforms. While the specific details of individual systems vary, most follow the general synthesis flow that is outlined in Figure 3.

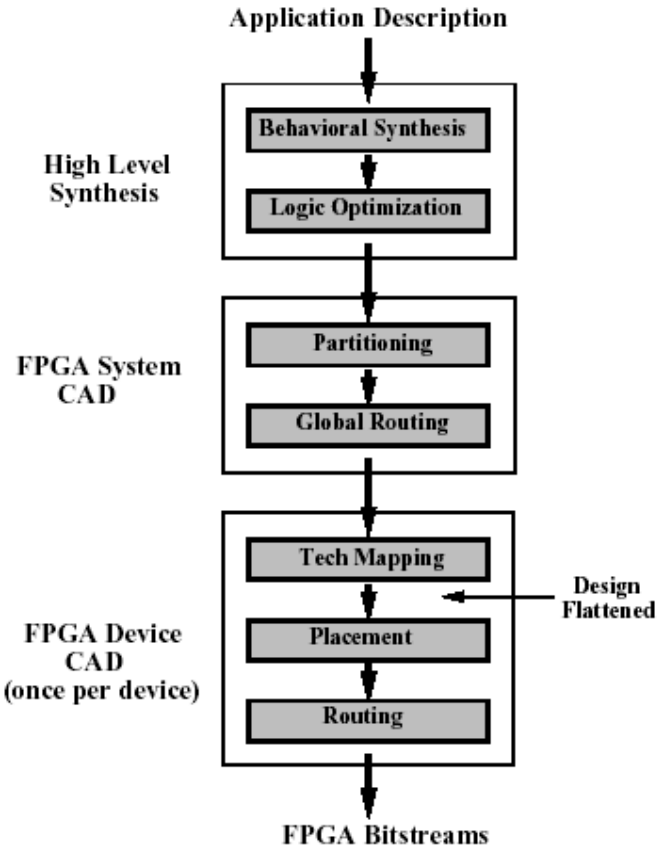


Fig 3.3. Reconfigurable Computing Synthesis Flow

High Level Synthesis

A user algorithm is typically specified in a high-level language (such as C or C++) or in a behavioral hardware description language (VHDL or Verilog). This representation not

only serves as a basis for synthesis but also can be simulated on a microprocessor for verification. Unlike microprocessor systems which require conversion of the textual representation to a sequence of simple processor instructions, reconfigurable computing systems require the generation of a complete hardware circuit. This synthesis step typically requires the *allocation* of datapath hardware resources in the form of high-level blocks such as ALUs, multipliers, and memory components, and the *scheduling* of communication between these components. Control of scheduled communication is maintained through the creation of control circuitry. This set of datapath and control structures form a register transfer level (RTL) representation of the application.

In the next step of the translation process, portions of the design in the control structure and datapath are optimized to a minimized set of Boolean logic gates through logic optimization. Frequently, this optimization is the same for FPGAs as for other VLSI technologies, such as full-custom design, and involves evaluation of issues such as required design performance and available circuit area. The result of logic optimization is a structural netlist of gate-level components grouped within the coarse-grained datapath macro-blocks defined by high-level synthesis.

FPGA System CAD Flow

Following creation of a macro-based circuit representing application behavior, the circuit must be mapped to a hardware system consisting of multiple FPGA devices. The steps by which a specific reconfigurable computing software system performs this translation process varies somewhat from system to system but in general the macro-based netlist created by high-level synthesis must be *partitioned* into smaller netlists for each FPGA device and inter-FPGA signals must be globally *routed* using system-level routing resources. In the partitioning step, the netlist generated by logic optimization is subdivided into pieces of circuitry small enough to meet the logic and inter-chip communication capacities of the target FPGA devices. Inter-FPGA connections are assigned to specific pins on the FPGA device and inter-FPGA signals are routed using system-level routing resources.

FPGA Device CAD Flow

In the technology mapping step of FPGA compilation, the functionality of primitive logic gates, generated during logic optimization, is restructured into sets of these basic blocks. If a primitive gate has more inputs that a single lookup table, its functionality

must be spread across several LUTs. Alternatively, if primitive gates contain too few inputs, small numbers of gates may be clustered together into groups for translation. After technology mapping, all design logic has been mapped into logic blocks at the quantization level of the basic block of the island-style device. The next step in the translation process is to assign the packed blocks of logic to speci.c logic block locations in the prefabricated two-dimensional array. The goal of placement for island-style FPGAs is to create a placed configuration of logic blocks that can be successfully interconnected in a subsequent routing step given the routing resources available. Routing is the process of identifying exactly which routing segments and switches should be used to create connected paths from net sources to net destinations for all nets in a circuit. Routing for FPGAs is complicated by the fact that the amount of routing resources in the FPGA device is fixed. In general, routing resources in non-congested portions of the device will be wasted while resource overuse in congested parts may lead to failure to achieve a successful route.

Design Flow Summary

Of the tasks listed in Figure 2, 90% of the compilation time for reconfigurable computing systems is typically spent performing FPGA place and route. This is due to the fact that while the other steps in the synthesis process typically optimize at the macro-block level, annealed placement of individual logic designs takes place at the grain size of the primitive logic blocks of the device. Not only does this approach require the placement tool to reconstruct locality information for a design which may contain high-level structure each time placement is performed, but also requires that all nets in the design be routed from scratch each time.

The cryptography algorithms discussed in Chapter II, along with their symmetric and asymmetric counterparts, have several implementation platforms. Broadly speaking, these platforms can either be hardware or software. On hardware, the classical implementation has been on ASIC (Application Specific Integrated Circuits), but recently, reconfigurable platforms such as Field Programmable Gate Arrays have been gaining in popularity. However, the easy solution is to implement the algorithms on software. In this case, the code might be meant for an Intel type or RISC type general purpose processor, or it might be in a constrained environment such as an embedded microprocessor or microcontroller in a DSP or smart card, for example. This is brought out pictorially in Fig. 4.1.

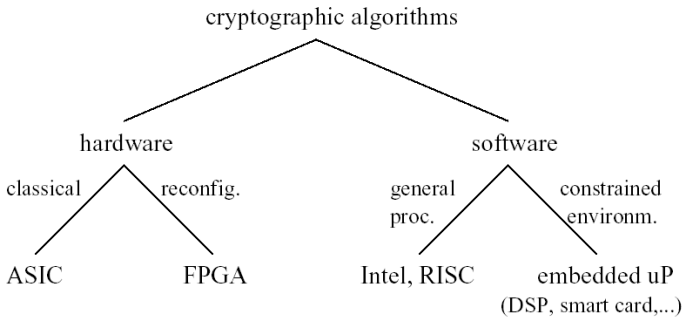


Fig. 4.1 Implementation Platforms for Cryptography Algorithms

4.1 Motivation for Cryptographic Hardware

The choice of the implementation platform is dependent on a variety of factors. Important amongst them are the required performance of the algorithm which is to be implemented, the capital available in terms of per unit as well as development cost, power consumption factors (very important in case of wireless devices!) and the flexibility of the implementation in terms of changes in parameters, key agility and algorithm agility, and the importance of physical security of the implementation itself. In total, we can not say that a particular platform is best suited for the implementation, because, it depends heavily on the requirements.

Fig. 4.2 compares the platform characteristics of the three major classes of implementation, namely, hardware, software and reconfigurable hardware. It can be determined from the comparison that reconfigurable platforms such as FPGAs manage to combine the advantages of hardware and software implementations.

On comparing the hardware and software implementations for cryptographic primitives, it can be determined that there exists some applications for which the software implementations are too slow. Further, there are also applications where physical security is of paramount importance. Hardware implementations are intrinsically more physically secure, since key access and algorithm modification is considerably harder.

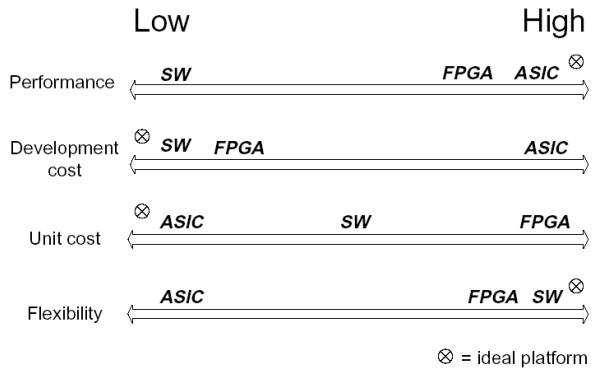


Fig. 4.2 Comparison of Platform Characteristics

In particular, crypto algorithms implemented on reconfigurable platforms offer a host of advantages. These include algorithm agility, algorithm uploadability, architecture efficiency, resource efficiency, algorithm modifications, increased throughput (relative to software) and cost efficiency (relative to ASICs).

Algorithm agility refers to the fact that modern day security protocols are defined to be algorithm independent. The algorithm is negotiated on a per session basis and a wide variety of ciphers can be required. Future extensions of the algorithm will also be allowed. Unfortunately, ASICs can provide algorithm agility only at a high cost. Algorithm upgradability implies that an application might require an upgrade to a new algorithm due to various reasons (breakage of current algorithm, expiration of standard, creation of new standard, extension of algorithm list of algorithm independent protocols. Upgrading of ASIC implemented algorithms are practically infeasible if many devices are affected or if the area of application is satellite communication and the like. Further, reconfigurable computing may allow architectures optimized for specific algorithm instances. Using runtime reconfiguration, the same FPGA can be used for both private key as well as public key algorithms in one session because a majority of the security

protocols do not use both of them simultaneously, which improves resource efficiency manifold. Algorithm modification can be readily implemented on FPGAs if there is a need, as in the case of cryptanalytic hardware. Hence, it comes as no surprise that there is no dearth of research into reconfigurable platforms for implementing cryptography algorithms.

Chapter V A Unified Architecture for Cryptographic Hash Algorithms

The study of various cryptographic algorithms and the hardware architectures for them, along with the knowledge of the latest research trends in the area enabled the development of an unified architecture for three hash algorithms covered in detail in Chapter II, namely, MD-5, SHA-2 and RIPEMD-160. Present day cryptographic coprocessors can implement multiple algorithms, but, unfortunately, they are not reconfigurable. The present implementation is for three different hash algorithms on the same FPGA which provides much more flexibility for the end user, as well as reconfiguration options.

The Previous work already done in this field is the integration of RIPEMD-160 , MD-5 and SHA-1. Here I propose the changes in the architecture to design another hash-chip with RIPEMD-160 , MD-5 and SHA-2 and then the extension of the same 32-bit chip to encompass a 64-bit algorithm Tiger.

Before proceeding with the description of the proposed architecture, the guidelines which were followed [12] in designing it are presented below.

5.1 Digital System Design Guidelines

Designing the hardware for the cryptography algorithms described in Chapter II is undoubtedly a question of properly following the digital system design process. Given beneath is the broad framework in which the design was carried out.

- Logic design is not the same as Verilog coding. One common mistake of some inexperience logic designers is to treat logic design as a Verilog programming task. This often results in Verilog code that is hard to understand, hard to implement, and hard to debug. Logic design is a process where one needs to:
    - Understand the problem.
    - If necessary, divide the problem into multiple modules with clean and well defined interfaces.
    - For each module:
        - Design the datapath that can process the data for that module.

        - Design the controller to control the datapath and produce control outputs (if any) to other adjacent modules.

    Verilog coding, on the other hand, is a modeling task. More specifically, after one has done some preliminary designs on the datapaths and controllers, Verilog code is then used to:

    - Model the datapaths and the controllers.
    - Connect the datapath and controller together to form modules.
    - Connect the modules together to form the final design.

- The design MUST be as simple as possible and easy to understand! If a design is hard to understand, then nobody will be able to help the original designer with his or her work. Also as time passes, the hard to understand design will become impossible to maintain and debug even for the original designer. Therefore, a logic designer must keep his or her design simple and easy to understand even if that means the design is slightly bigger or slightly slower as long as the design is still small enough and fast enough to meet the specification.
- Use an hierarchal strategy that breaks the design into modules that consists of datapaths and controllers. More specifically:
    - Divide the problem into multiple modules with clean and well defined interface.
    - For each module:
        - Design the datapath that can process the data for that module.
        - Design the controller to control the datapath and produce control outputs (if any) to other adjacent modules.
- Keep different clock domains separate and have an explicit synchronization module for signals that cross the clock domain.
- The best way to study the effect of the datapath's pipeline registers is to draw a timing diagram showing each register's effect on its outputs with respect to rising or falling edge of the register's input clock.
- The block diagram of the datapath should show ALL registers, including the implicit register of the Sequential Datapath Element.
- While designing the Sequential Datapath Elements, separates the element into the two parts:
    - The combinational logic

    - The register.
- The best way to decide when and where to use pipeline register or registers to stage the controller inputs and outputs is to draw a timing diagram showing each register's effect on its outputs with respect to rising or falling edge of the register's input clock.
- The block diagram of the controller should show ALL registers explicitly while the random logic can be represented by a simple black box.
- If possible, use one-hot encoding for the finite state machine's state encoding to simplify the Output Logic as well as the Next State Logic.
- Instead of designing a controller with a giant and complex finite state machine at its core, it may be easier to break the controller into multiple smaller controllers, each with a smaller and simplier finite state machine at its core.
- For finite state machine design, keep the Next State Logic block separate from the Output Logic block.
- In a Mealy Machine design, it is possible to use the Next State Logic block's output as inputs to the Output Logic block. This must be done with caution since the total delay of the two logic block may become the critical path of the controller.
- A finite state machine containing states whose transition to their next states are governed only by the number of cycles it has to wait can be simplified by building a multiplexer tree to select the number of cycles a counter must count before generating an "expire" signal to trigger the state transition.

5.2 Hash Algorithm Characteristics

This section presents the similarities and dissimilarities between MD-5, SHA-256 and RIPEMD-160 which had a major influence on the design of the architecture for the algorithms.

- While MD-5 has a 128 bit message digest output, SHA-256 has 256 bit and RIPEMD-160 have a 160 bit message digest output. Hence, it is necessary to have extra 32 bit ‘E’ registers in the CVQ block, as well as the intermediate register sets.

- Even though SHA-256 and RIPEMD-160 require very few constants for their operation, MD-5 requires a table of 64 values. Hence, it was decided to store all the constants in a ROM.
- Special care had to be taken in the padding block, because the padding of the count for SHA-256 had to be in the big endian format, while it was in the little endian format for MD-5 and RIPEMD-160.
- SHA-256 has a very complex method of taking inputs to the elementary step from the message blocks. While MD-5 and RIPEMD-160 choose the message words directly as inputs to the compression function, SHA-256 calculates 64 different inputs from the given 16 words. This required an additional register file especially dedicated to storing the SHA-256 processed message words for future selections.
- MD-5 and SHA-256 have only one round executing at a time. However, RIPEMD-160 has two rounds executing in parallel, as was presented in Chapter II. Hence, there is a necessity to duplicate some of the resources in a parallel block.
- Implementing three algorithms in the same chip has resulted in a very complex microcode unit for the architecture.

5.3 Datapath Components

The entire architecture is composed of a multitude of components, each of which is presented below.

- Fig. 5.1 illustrates a circular left shifter, which can perform shifting by variable shift amounts. In reality, it would be a barrel shifter after synthesis. DataIn is circular left shifted by ShiftAmount and passed on to DataOut.

Fig. 5.1 Circular Left Shifter

- Fig. 5.2 illustrates a 4:1 multiplexer, which transfers the appropriate input to the output depending on the values in the select lines.

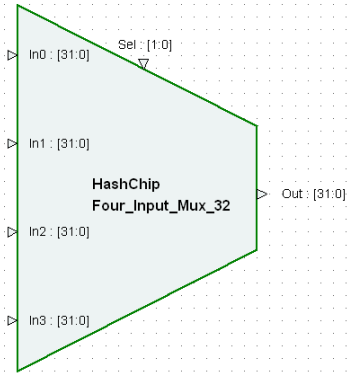


Fig. 5.2 4:1 Multiplexer

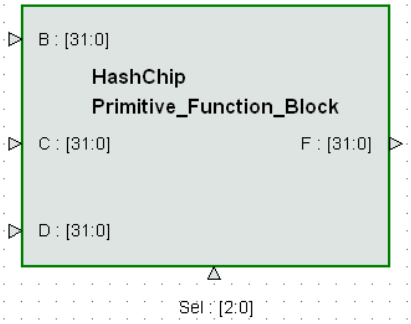


Fig. 5.3 Primitive Function Block

- Fig. 5.3 illustrates the primitive function block which generates a particular Boolean function (out of the 7 available) of the input variables depending on the values in the Select input.
- Fig. 5.4 illustrates the ROM Table block which supplies two constants to the two parallel datapaths depending on the values at the address pins.

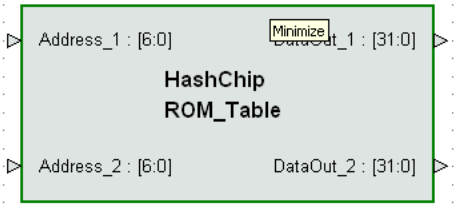


Fig. 5.4 ROM Table

- Fig. 5.5 illustrates the Register file which supplies the necessary message words during each iteration, for both the upper and lower datapaths.
- Fig. 5.6 illustrates the SHA Processing Block which performs the special operation done on the selected message words.
- Fig. 5.7 illustrates the SHA Register file which holds the results of the updation of the message words by the SHA Processing Block.

- Fig. 5.8 and Fig. 5.9 illustrate the structure of the two and three input adders respectively. The addition performed is modulo 2^{32}.

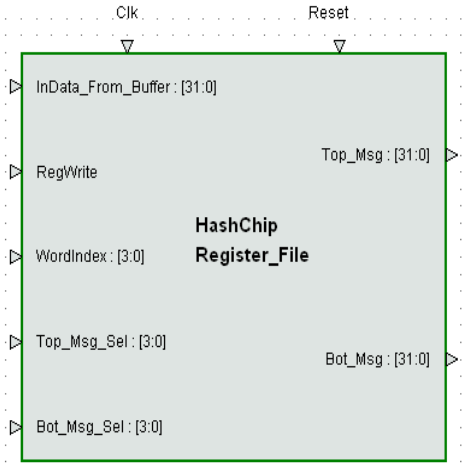


Fig. 5.5 Register File

- Fig. 5.10 shows a 2:1 multiplexer used at various places in the datapath as shown later.

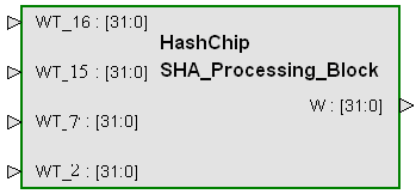


Fig. 5.6 SHA Processing Block

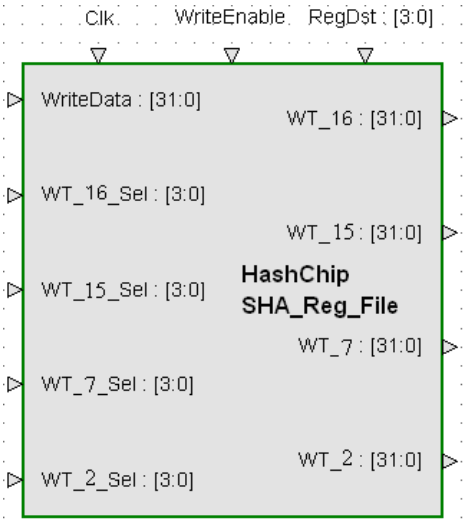


Fig. 5.7 SHA Register File

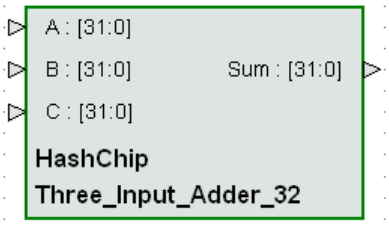


Fig. 5.8 Three Input Adder

Fig. 5.9 Two Input Adder

- Fig. 5.11 illustrates the signals of the Padding Block cum FSM Controller. The padding block determines when the message actually stops, and does the padding of a '1' bit followed by '0' bits, finally ending with the padding of the count in the proper format (big or little endian). The state machine from which it is designed is presented later.
- Fig. 5.12 illustrates the signals of the microcode control unit, which is used to **control the operation of the datapath in each and every cycle.**

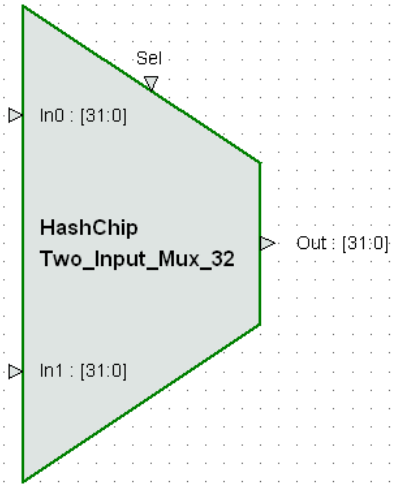


Fig. 5.10 2:1 Multiplexer

Fig. 5.11 Padding Block and FSM Controller

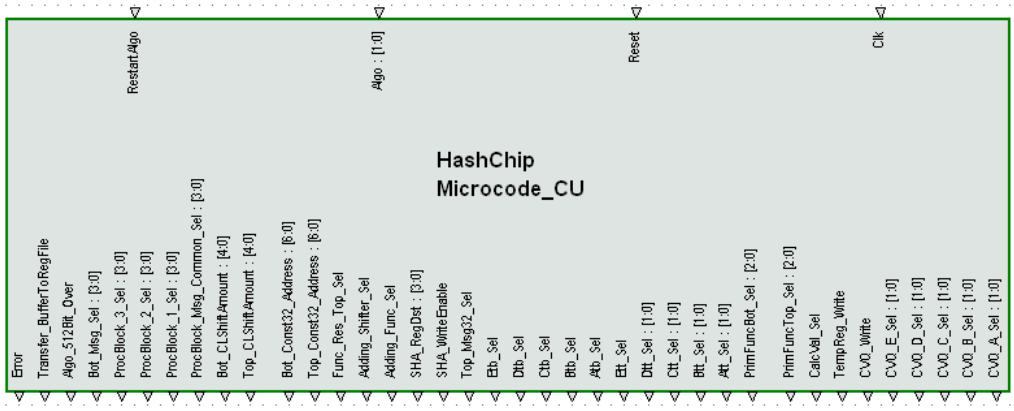


Fig. 5.12 Microcode Control Unit

5.4 Implementation and Complete Datapath

In the above list of datapath components, everything except the Padding Block / FSM Controller and the Microcode Control Unit is straightforward to implement. In this section, the FSM for the Padding Block and FSM Controller is presented, along with the details of the microcode control unit and the complete datapath of the proposed architecture.

In constructing the FSM for the padding block, it is assumed that the chip would be interfaced with a memory bank and a memory controller in which a Read signal would be acknowledged with a single byte of data. Further, the message input to the algorithm is assumed to be having a length which is a multiple of 8 bits. The message ends when the input data from the memory bank is 00H. The other details of the FSM can be obtained by referring to the code discussed in the appendix. Fig. 5.13 presents the FSM for the padding block.

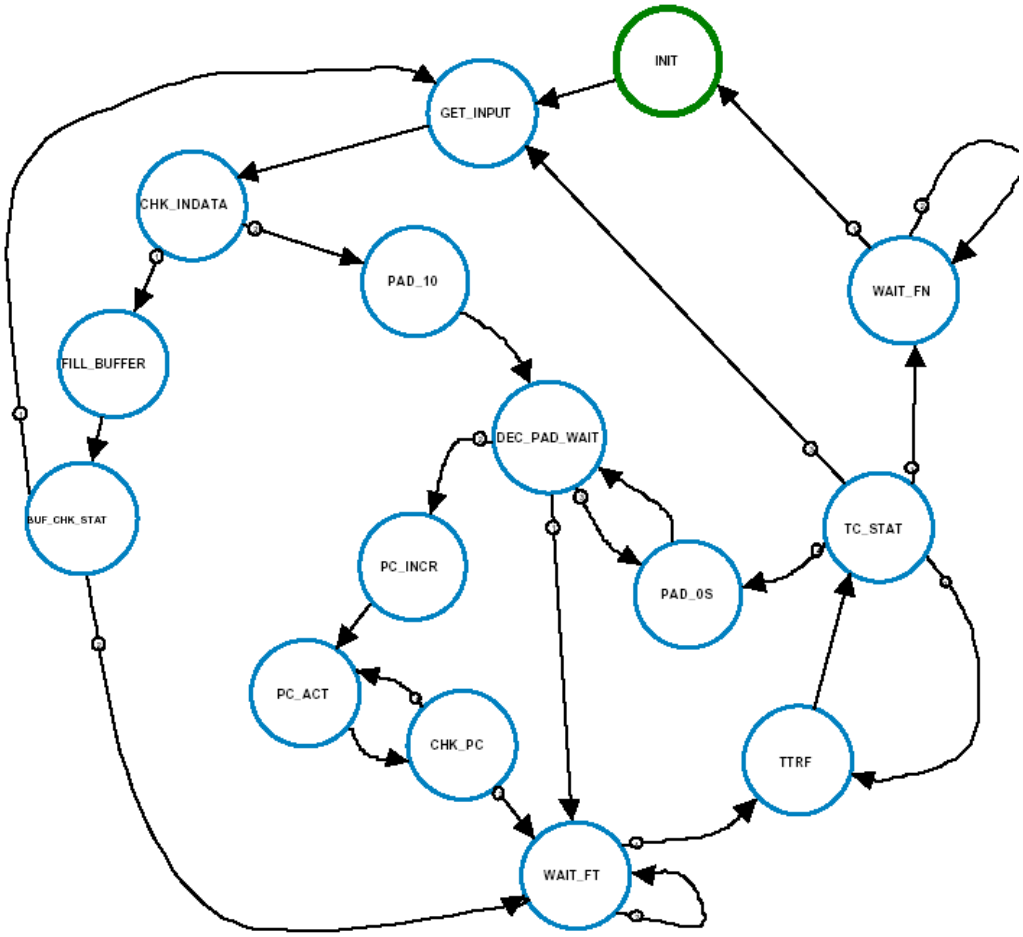


Fig. 5.13 FSM for the Padding Block Controller

The microcode control unit is envisaged as a ROM table of values, which feed the control signals at various points of execution. To be more precise, the datapath components are controlled by values which change with the change in the round and step of each of the algorithm. Thus, for every single step of each of the algorithms, there is a control unit entry. In addition, we have entries for invalid inputs, reset states etc. The present complexity of the designed control unit is 235 lines of 93 bits each! This would change with the updating of the design to include reconfiguration ability and more algorithms

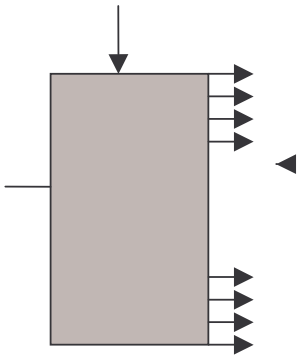
5.5 Extension of the Chip to include Tiger as well.

On comparing Tiger with the rest of the algorithms. it can be seen that the structure of tiger is very different from the others and as such including 64-bit Tiger on the 32-bit architecture with so much dissimilarities is indeed a challenging task.

The only similarity is the padding block .So some new blocks had to be introduced

Which are described as under :-

S-Block: The Original four 8 X 64 S-boxes are implemented using eight 8 X 32 S-boxes which is then duplicated.



SBox Block

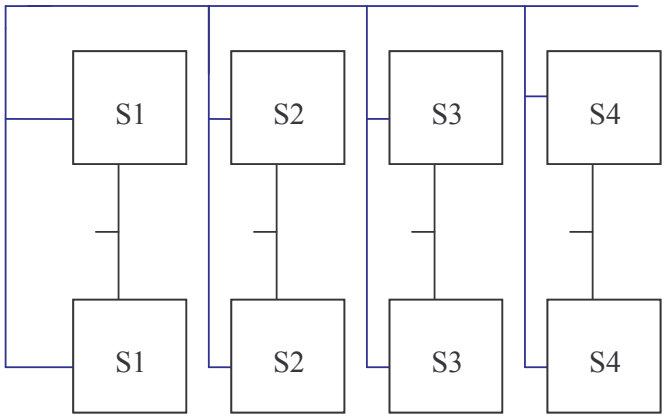


Fig 5.5.1 Upper S-Box Block (64 bit words stored in two 32-bit words)

The Upper S-Box block gets First 32-bits of C i.e the bytes C_0 , C_1 , C_2 and C_3 and as such is capable of performing the lookups pertaining to these bytes. (t1[C_0] and t2[c_2] for A and t4[C_1] and t3[C_3] for B). The four 64 bit words so obtained come out as eight 32-bit lines from the box.

The Lower S-box gets the Last 32-bits of C and generates the t3[C_4],t4[C_6] parts for A and t2[C_5], t1[C_7] parts for B. All these lines are then sent to Tiger Processing Block.

Tiger processing Block

Tiger processing block internally consists of some XOR blocks , Adder and a Subtarctor.

Fig 5.5.2 illustrates the complementing block that consists of 32 two-input XOR gates that function as programmable NOT gates, complementing the input data when control input is 1.

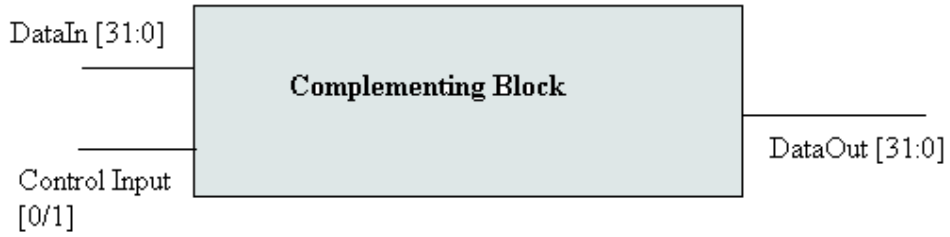


Fig 5.5.2 Complementing Block

Fig 5.5.3 Adder/Subtractor – 64 bit

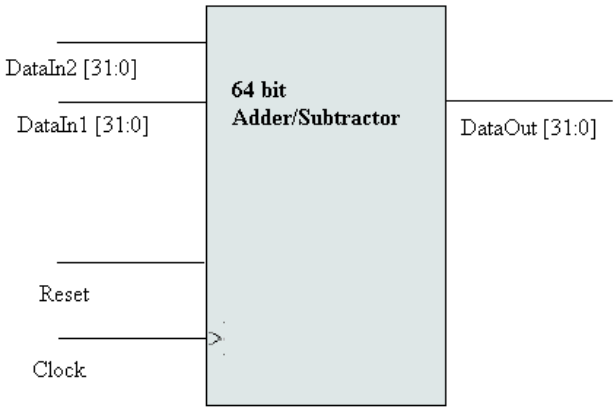


Fig 5.5.3 illustrates a 64 bit adder/subtractor unit which performs the 64bit operation in two clock cycle. It consists of a 2 input adder , a 3 input adder, a 2:1 multiplexer and two D Flip Flops. The unit is reset to zero when performing addition and set to 1 when performing subtraction. The first input of the 2-bit adder (DataIn1) is a 32 bit input. The second input comes from a D flip flop which has been set to one or zero depending on the operation (addition or subtraction) to be performed. This D flip flop also stores the carry output of the 2 bit adder. The sum output of the 2-bit adder goes to the first input of the 3input adder.The second input is DataIn2. The third input comes from a 2:1 multiplexer whose inputs are Zero or the carry stored in the D Flip Flop depending on the whether the operation is performed on the upper or lower 32 bits. The final DataOut is the sum output of the 3-input adder.

Select Round Pass Block

This block selects what pass and round it is and changes the order of the parameters accordingly. Internally it consists of some Muxes only.

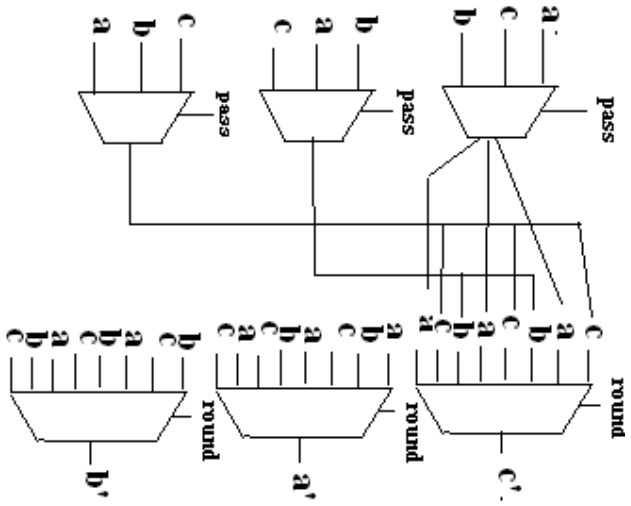


Fig 5.5.4 Pass/Round Select Block

Again we have two such blocks , each taking just 32-bit a,b,c as input and generating the corresponding first or last 32-bits of the word.

Apart from these some other blocks to be used include a special register file for Tiger **"Tiger Reg file**" with individual read and write enable signals for every register.
It contains the X0-X7 registers (16 X 32) , a ,b,c (6 X 32) and aa,bb,cc (6 X 32) registers. Hence the total size of the Reg File will be 28 X 32.

Other blocks include some decoder / demuxes and a couple of Shifters.

Fig 5.5.5 illustrates a 64-bit Shifter Block that can shift left and right depending on the control input. In case of MD-5, SHA-256 and RIPEMD-160, the lower 32 bits of DataIn and DataOut are used.

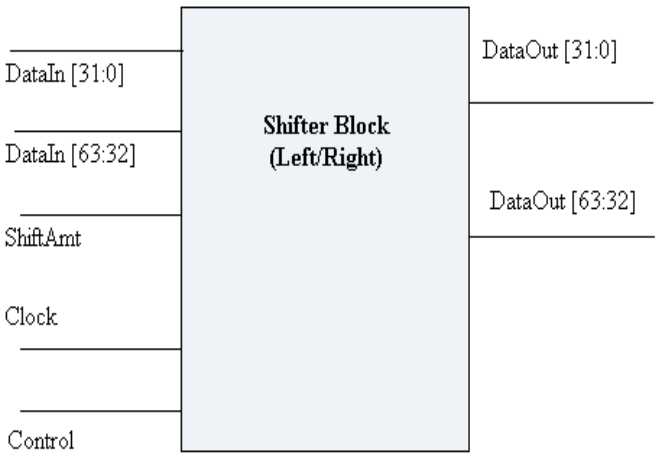


Fig 5.5.5 Shifter Block

Key Schedule Block

Again two such blocks have to be used.

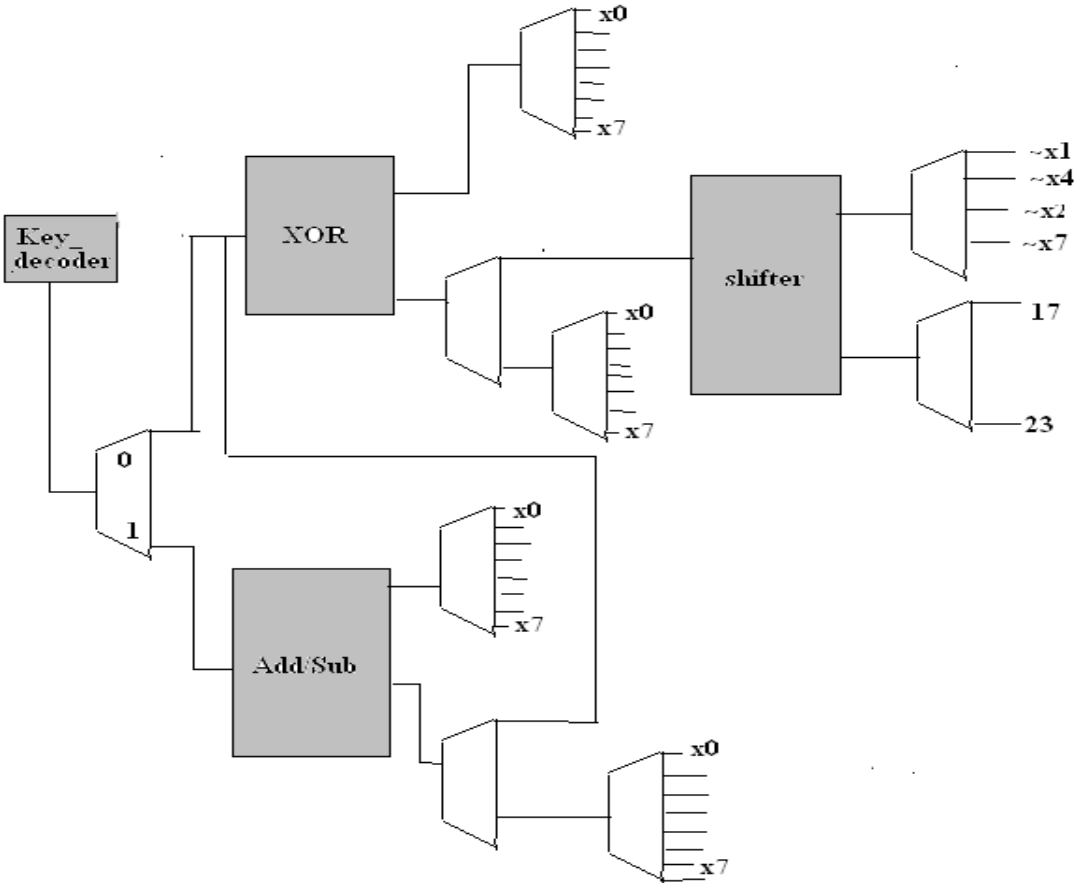


Fig 5.5.6 Key Schedule Block.

5.6 Verilog RTL Coding Guidelines

It is of paramount importance to ensure that the HDL code which is being written for the purpose of implementing the architecture on an FPGA be synthesizable. There are quite a number of guidelines to be followed [13], important ones of which are presented below.

- Draw a simple block diagram, labeling all signals, widths etc.
- Draw a timing diagram with as much detail as possible
- Code the HDL according to the synthesizable templates
- Do a quick, low effort, compile- just to see if it is synthesizable *before* simulating. Compare this to the block diagram. Look at the inference report:
    - Count the number of flip flops - is it the same as the number of flip flops in the code.
    - Check for latches - did you want them. If not, latches are inferred in combinational procedures - the inference report tells you which combinational procedure and the name of the latch. Fully specify all variables in all cases to eliminate latches.
- Check the case statement inference. Was it full/parallel?
- Check any incomplete event list warnings?
- Check to see if there are any combinational feedback loops (typically only after a compile). Combinational feedback loops can be identified by the signal names in the timing loop.
- Check the schematic - any ports unconnected?
- Never ignore any warning that the synthesis tool flags. All warnings need to be understood and typically signed off.
- Simulate and compare with the timing diagram

Coding State Machines

- Draw a state diagram. Label the states. Allocate state encoding.
- Label the transition conditions and label the output values.
- Use parameters for the state variables.
- Use two procedures (one clocked for the state register and one combinational for the next state logic and the output decode logic).
- Use a case statement in the combinational procedure.
- Have a reset strategy (asynchronous or synchronous).
- Use default assignments and then corner cases.
- Keep state machine code separate from other code (i.e. don't mix other logic in with the state machine clocked and combinational procedures).

Chapter VI Conclusion & Future Work

Starting from the ground up, a new architecture has been proposed as a unified solution for three different cryptography algorithms. (RIPEMD-160 , MD-5 and SHA-256) The architecture has been coded using synthesizable Verilog RTL constructs, and all the components have been tested for synthesizability on a variety of FPGAs. Testbenches have been written for each of the modules and their functioning verified. There is much scope for improvement, though. The following are guidelines for future work based on this thesis

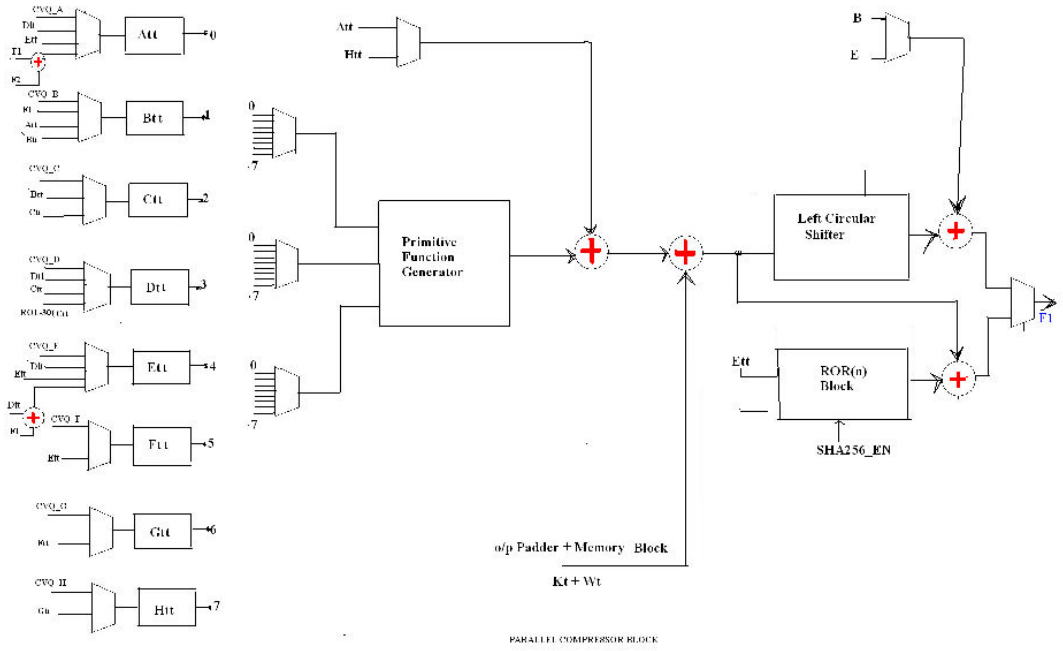
- Integrate all modules together to make the complete system
- Write out testbenches which exhaustively cover out all possible cases
- Perform the very important step in the design flow, namely, back annotation, wherein the final netlist is again simulated to check for pre synthesis and post synthesis simulation mismatches
- Integrate more hash algorithms into the chip. While this should prove easy, implementing a cryptography coprocessor like system on a reconfigurable fabric would prove to be challenging.

The Proposed datapath along with Tiger integration has to be coded and tested exhaustively.

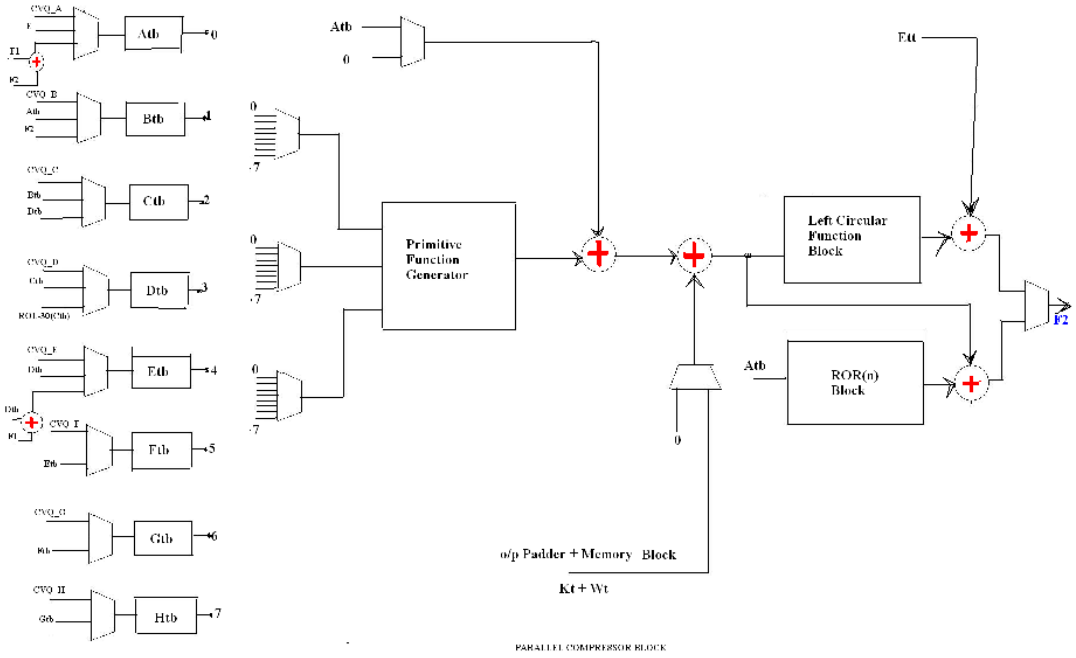
Appendix A Coding Details

The code for implementing the architecture has been written in Verilog HDL. Model Technologies' ModelSim v5.7G was used extensively for simulation purposes. For synthesis, Exemplar Logic's LeonardoSpectrum v2001.3 was used along with Synplicity Synplify Pro v7.3.3. The details of the code are as presented below:

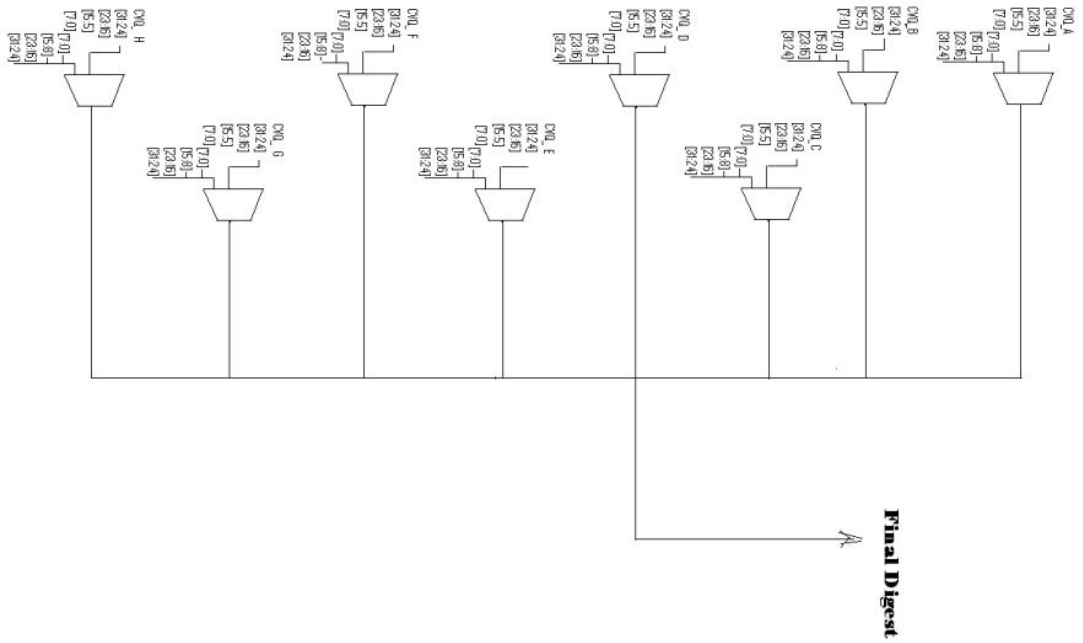
- Adder.V – Contains modules of two input and three input 32 bit adders which perform modulo 2^{32} addition.
- CLShifter.V – Contains the circular left shift module
- Multiplexer.V – Contains the descriptions of 32 bit 2:1 and 4:1 multiplexers
- PadBlock.V – Contains the Padding_Block_FSM_Controller module
- PrimFunc.V – Contains the implementation of the seven primitive functions in the Primitive_Function_Block module
- RegFile.V – Contains the Register_File module which holds the message words in terms of 16 32 bit words
- ROMTable.V – Contains the ROM_Table module which is basically a dual port ROM implementation for the constants required in the algorithms
- SHABlock.V – Contains the module to perform the operation required before the message words can be used inside the SHA algorithm itself.
- SHARegFile.V – Contains the register file required by SHA which computes 80 different words from the 16 message words available to it in the original register file.
- Microcode.V – Contains the microprogrammed control store for the system in the Microcode_CU module
- HashChip.V – Contains the top level interconnections to build the whole system



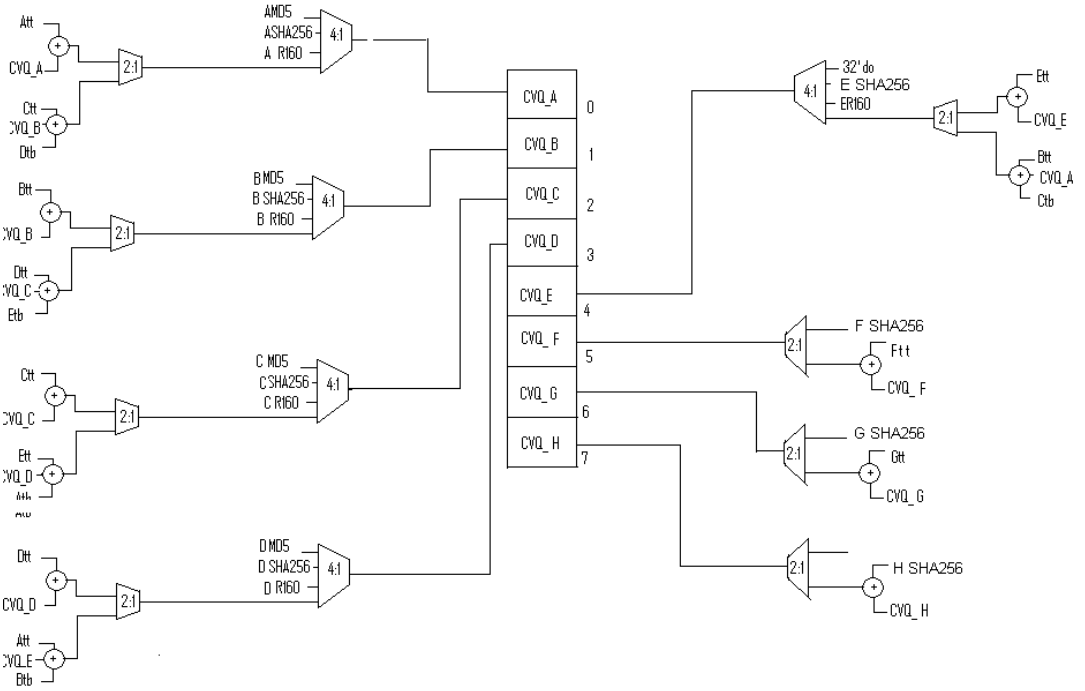
The Main Compressor Block



Parallel Compressor block



Digest Generation Block



CV-Updation Block

INSTATLLATION AND SIMULATION

In order to run the simulation, we need to have ModelSim 5.7G or higher installed

on our computer. Assuming the files of the release are copied into a directory on the computer, say, C:\chip\.

Assuming that the extraction was performed into C:\Rohit\HashChip\PostMod, the directories inside will be having their full path as C:\Rohit\HashChip\PostMod

On Starting ModelSim 5.7G, go the File Menu and click on 'Close' in order to close the current project. After this, in the main window, type in the following commands at the 'ModelSim>' command prompt as shown below,

```
ModelSim> cd C:/Rohit/HashChip/PostMod
ModelSim> do HashChip.do
```

This will start the execution of the macro written for the creation of the project and its compilation. The Tcl/Tk script written for creating a GUI for the simulation is also interpreted and executed.

A brief guide to the simulation GUI is given here. The layout of the GUI is as shown in Fig. 01. The main window is a non resizable one of geometry (800 X 600). It has got a pane for displaying copyright and author information at the top. Immediately below this are two panes, the left one containing radio buttons for choosing the particular algorithm to simulate and the right one containing a text box and a button for choosing the file to hash. The area below this is also split into two panes, the left one for displaying all information which have been coded directly into the TestBench module, such as the clock period, maximum file size which can be hashed, and the file into which all the simulation results are written. The exact details which are required to be written into the log file can be modified by setting or clearing particular bits in the TestBench module. Further details can be had from the comments provided in TestBench.V. Also, information about the range of microcode unit addresses for each algorithm is provided in the same pane.

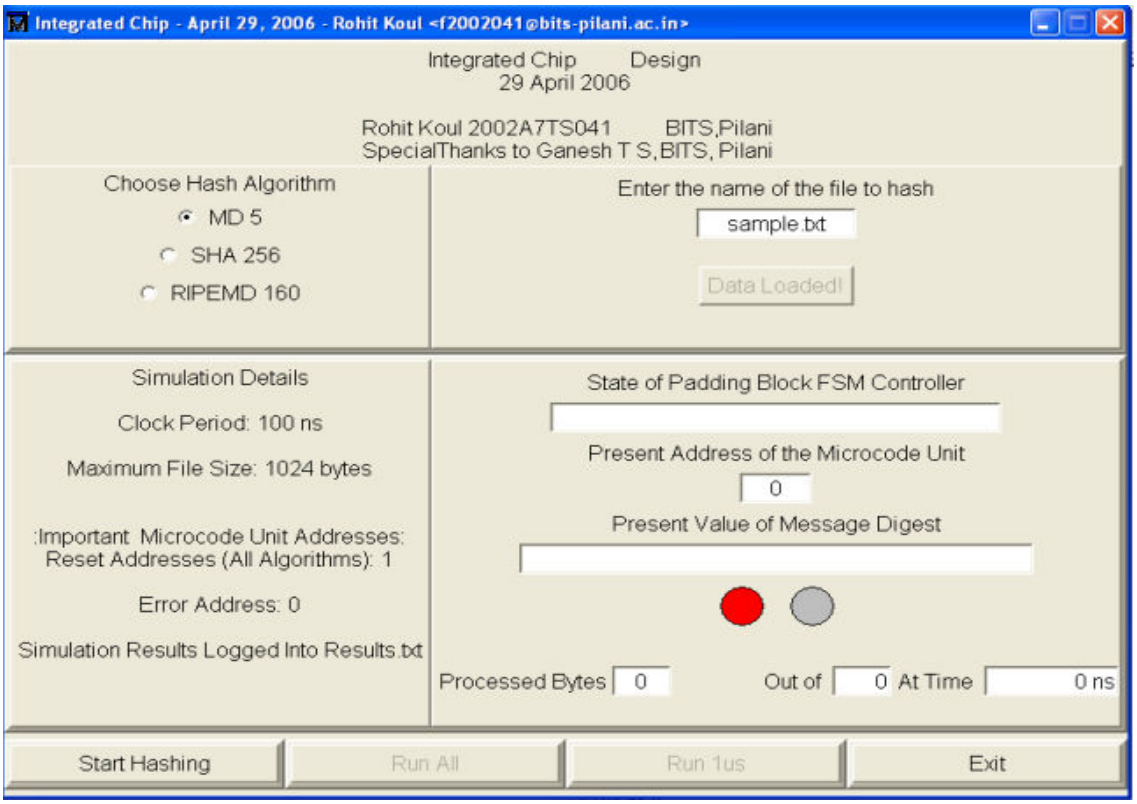
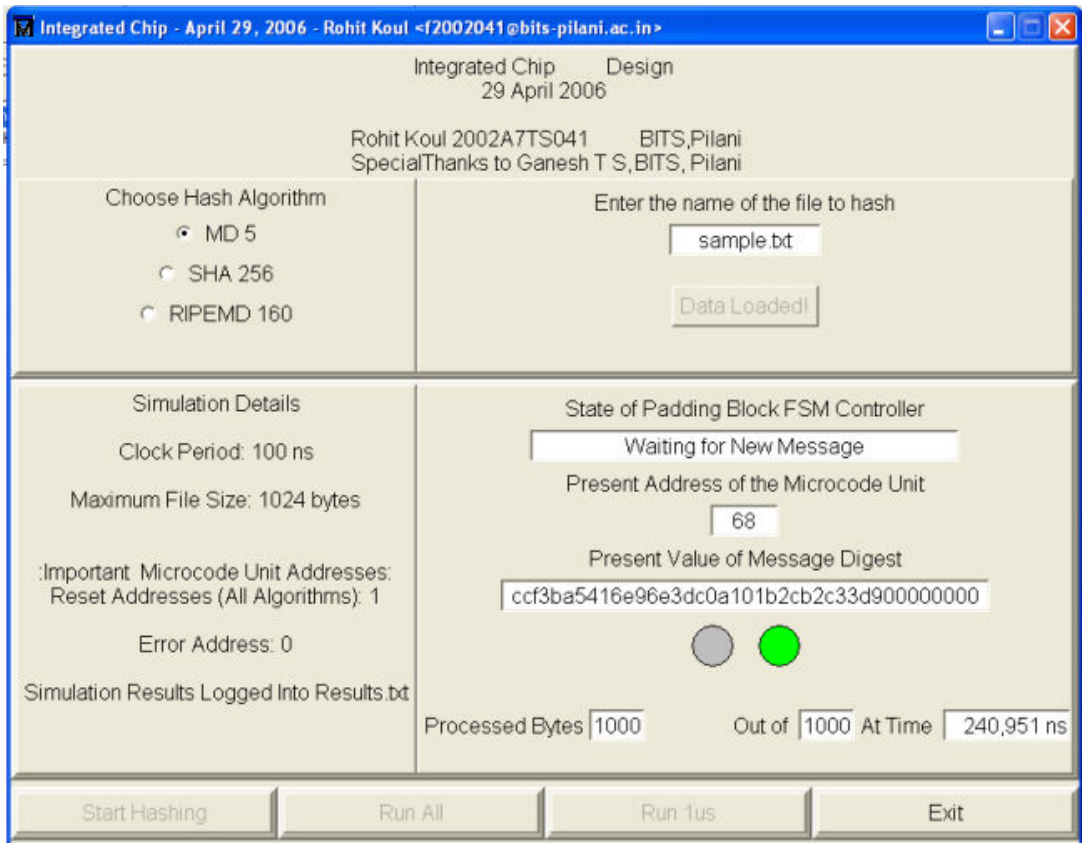


Fig. 01 GUI for HashChip Simulation - Sample Screenshot

The pane immediately to the right gives the values of the most important variables during the simulation. It displays the current state of the FSM controller of the padding block, followed by the address of the microcode unit control signals which are being currently executed. Then, the most important of them all, the value of the message digest at the present simulation time, is displayed. Below this textbox are two ‘signals’, which indicate the validity of the message digest displayed just above it. If the red signal is active, it means that the digest is yet to be computed fully. Once this gets grayed out and the green signal turns on (as it is in Fig. 01), it means that the message digest displayed is a valid one, obtained after considering all iterations for all the message bytes. Below these signals, we have information about the processing status of the message to be hashed. It displays how many bytes have so far been taken into the intermediate registers out of the total size of the file. It also displays the current simulation time.

At the bottom of the window, we have buttons to control the simulation. Options include beginning the simulation, running it till a valid message digest is generated, running it for 1000ns (ten cycles, according to the present setting of the clock period



in the test bench), and also to exit the simulation. Presented next is details of how to go through the simulation.

After the GUI starts, choose the algorithm to simulate, by clicking on the required radio button. It is set to MD5 by default. Follow this up by typing in the name of the file to hash. It is Sample1.txt by default. Please note that it is necessary to limit the size of the file to less than 1KB (can be altered by changing the necessary parameter in MemBank.V).

Also , we need to make sure that, if we are going to choose a different file to hash, it is in the 'postMod \' directory only. Click on the 'Load Data' button in order to convert it into a form readable by the Verilog simulator. Once this has been done, the buttons at the bottom of the window become active. Choose 'Start Hashing' in order to initialize the simulation, reset the HashChip and load the file data into the internal memory bank. We can then follow up the simulation by either clicking on the 'Run All' button or 'Run 1us' button. To provide an idea of how many cycles are taken for simulation, an

empty message takes around 240 cycles (24us at the present clock rate) to get processed and give out the digest, while it is 1140 cycles for a 206 byte message (as in Fig. 01) and 3490 cycles for a 740 byte message.Finally, We have an ‘Exit’ button in order to quit the simulation and return to the ‘ModelSim>’ prompt in the main window.

BIBLIOGRAPHY / REFERENCES

[1] Gary C. Kessler, Handbook of Local Area Networks. Cambridge, M.A; Auerbach Publications, 1998

[2] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography. Cambridge, M.A; CRC Press, 1996

[3] R. Rivest, RFC 1321. MIT Laboratory for Computer Science and RSA Data Security, Inc.; 1992

[4] William Stallings, Cryptography and Network Security – Principles and Practices 3rd Edition. New Delhi, India; Pearson Education (Singapore), 2003

[5] D. Eastlake and P. Jones, RFC 3174. Cisco Systems Inc.; 2001

[6] H. Dobbertin, A. Bosselaers, B. Preneel, RIPEMD-160, a strengthened version of RIPEMD. Fast Software Encryption, LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71-82.

[7] S. Brown and J. Rose, FPGA and CPLD Architectures: A Tutorial. IEEE Design & Test of Computers, vol. 13, no. 2, pp. 42-57, 1996

[8] Joan Deamen et al., A Hardware Design Model for Cryptography Algorithms. Katholieke Universiteit Leuven, Belgium, 1993

[9] R. Reed Taylor et al., A High Performance Flexible Architecture for Cryptography. CS Department, Carnegie Mellon University, 1996.

[10] Colin D. Walter, Systolic Modular Multiplication, PhD Thesis, University of Manchester, Manchester, England, Mar. 92

[11] A.J. Elbirt and C. Paar, Towards an FPGA Architecture Optimised for Public Key Algorithms, in The SPIE's Symposium on Voice, Video, and Communications, September 20, 1999, Boston, MA, USA

[12] Shing Kong (March 2001) Private Communication, archived at http://www.cs.wisc.edu/~markhill/kong